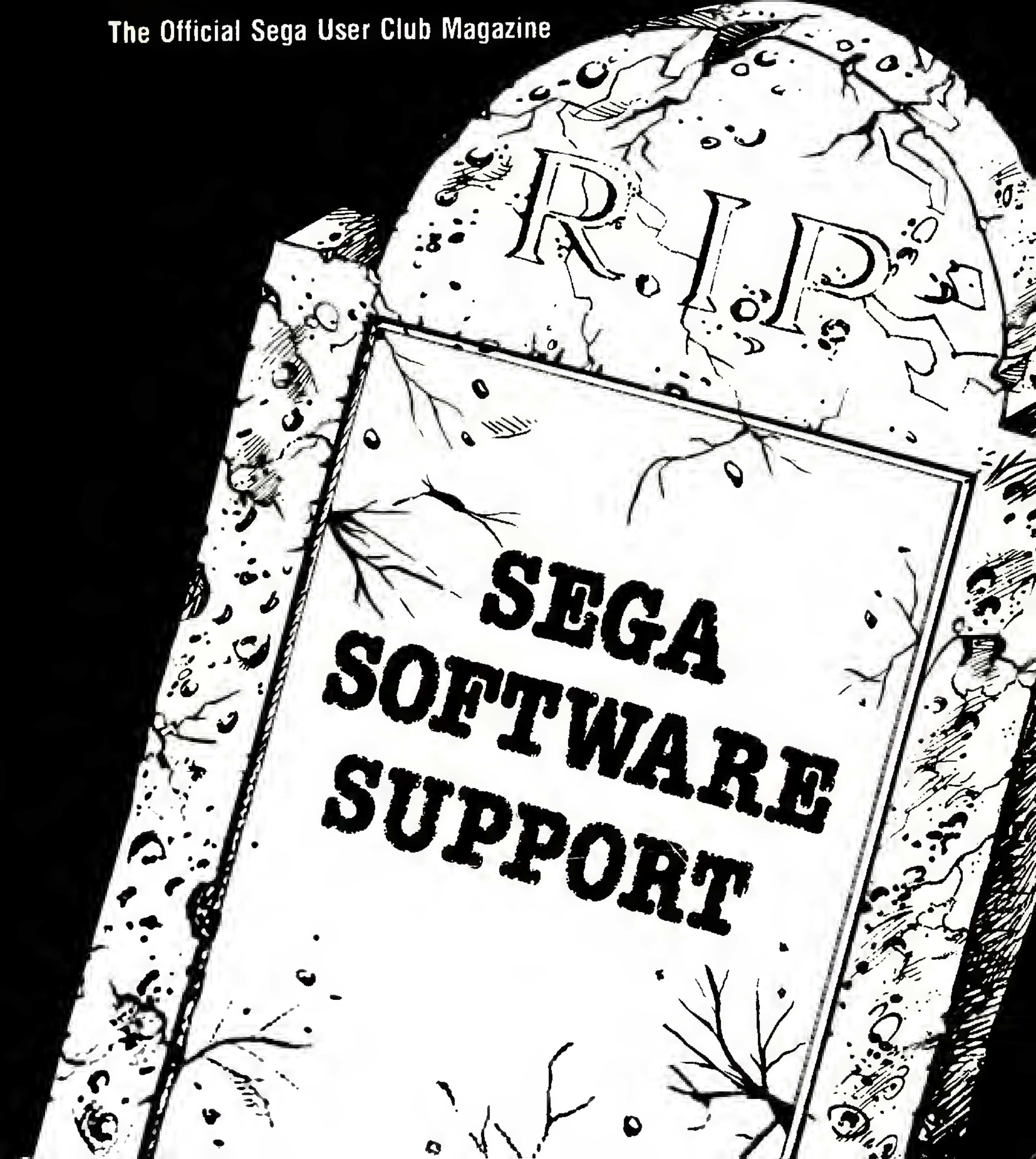
SEGA® COMPUTER

The Official Sega User Club Magazine

MARCH/JUNE 1987

- Sega Basic Revealed
- Easycode
- Parallel Resistors
- Sega Computer . . . final



SEGA®
COMPUTER

The official magazine of
the Sega User Club, New
Zealand.

CONTENTS

Registered as a magazine at the CPO Wellington, New Zealand.

EDITORIAL

EDITORIAL MAR/JUNE DOUBLE ISSUE

Due a change in my business direction Sega Software Support closed down on March 31.

This double issue of Sega Computer is the last to be published by Sega Software and completes the 1986-87 subscription year.

DO NOT DESPAIR!!!!

Geoff Crawford from Poseidon Software (a well known name in Sega circles) has taken over the operation from this date and all the current stocks sold by Sega Software will continue to be sold by Geoff, together with all the excellent programs produced by Poseidon.

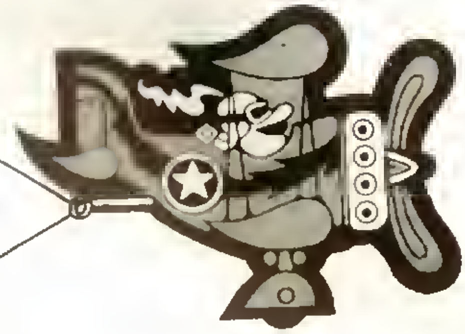
All enquiries and re-subscriptions, after this date, should be forwarded to Poseidon Software at P O Box 277 Tokoroa.

This FINAL issue from Sega Software includes almost every bit of information a Sega owner would want to know.

Contents include an article on Easycode (a simulated machine code for beginners) from `Computing Today', complete with programs adapted from the originals especially for the Sega. We have also reproduced parts of the disk drive manual, relevant to the operation of the Sega computer - not just for disk drive owners - it's all the information that was not included in the Basic Manual and is necessaary for using your Sega to it's fullest capacity. The final part of Brian Brown's Book `The Sega Computer' makes this issue a comprehensive manual in its own right.

The past year, since we took over from Grandstand, has been both rewarding and interesting and all of us at Sega Software hope you will give Geoff the same support as you have given us.

LETTERS...to THE EDITOR.



DEAR EDITOR

I would like to know the answer to a question which is:

(a) Why are original Sega games like Star Trek, Zaxon and Buck Rogers made for the Commodore computer but have never come out for the Sega computer?

(b) Are Flicky or Lode Runner ever likely to be in stock?

Stewart Parkes, Papakura.

EDITOR'S REPLY

Sega Japan can give us no reason.

DEAR EDITOR

I would like to thank you for your prompt response to my application to join your Sega User Club last year. I am very impressed with what I have seen of your magazine so far, it is way out in front of anything I have seen in Australia.

There is something I would like to mention if I could. The KARATE program featured in the Jul/Aug issue looked as though it would be tremendous, but it was a very big let down after hours of typing to find there were still gremlins in the program. The subsequent issue made no mention of any corrections and although I debugged it, I could still not get to run properly with a disk drive. Other local Sega owners have not been able to work out what was wrong either.

Andrew Alliston, Cardiff, NSW, Australia.

EDITOR'S REPLY

To run this program on a disk drive it is necessary to remove all the REM statements. To our knowledge there are no other bugs and the program runs perfectly on both tape and disk on our Sega.

DEAR EDITOR

(a) Regarding B C R Davis' problem with Jet Ranger, I had the same problem and found I had typing errors. Lines 10-20-30 with the incorrect number of 'As' and Data lines incorrect. If he still has problems he can send me a tape and I'll send him a copy.

(b) In the front of our magazine is a note about contributions being the origianl work of the author, but there are two programs by Jan Jacobsen in the Nov/Feb issue which were written by Tim Hartnell. Gomoku and Chess are from his book 'Giant Book of Computer Games'. Gomoku has been altered very slightly but Chess is an exact copy. There are strict copyrights on these programs and I don't think Jan should take credit for someone else's program. Could you please publish this letter in your next issue.to deter other people from doing the same.

(c) On page 11 Table XX14 Music program lines 250-630 seem to be missing.

(d) I think the mag is great but could we have a price list of software, etc. that's available from Sega Software Support, published with each magazine.

Terry Cole, P.O. Box 7140, Te Ngae, Rotorua

EDITOR'S REPLY

(b) As we had not heard of 'Giant Book of Computer Games' we were not aware that these programs were from this book. Agreed — copyright infringement is serious and we hereby withdraw the reference to Jan Jacobsen.

(c) Gremlins at work!!

(d) As you will know from the editorial, this is the last issue of Sega Computer being published by Sega Software Support — sorry.

DEAR EDITOR

(a) Can you or your readers give any information on how to retain titles or sub titles on the screen when using the Hucal disk eg. when listing down the rows the titles scroll up out of sight. By using the window function I was unahle to scroll with the columns.

(b) Also is it normal for the screen to change background colour after a few minutes when using disk programs? It changes from the normal program colorus to a grey background.

D. Mudgway, Feilding

EDITOR'S REPLY

(a) It is not possible to retain titles when scrolling down the screen nor to retain your scroll function when using the window function.

(b) Sound like you may have a problem with either the disk drive, computer or the TV set you are using.

DEAR EDITOR

Sega as an RTTY Terminal. I refer to the letter from J. Lindsay in the Nov 1986/Feb 1987 issue of the magazine in which he seeks information, software, and circuits for modems or interface to use the Sega as an RTTY terminal Presumably by RTTY Mr Lindsay is referring to a remote teleprinter, in which case I can assure Mr Lindsay that I have successfully operated my CREED Model 7b teleprinter from my Sega, writing a REM packed machine coded program for the purpose and using the printer socket as an output socket. I also use this socket for my latest printer. Before I can help Mr Lindsay (even though I will be temporarily resident in Bangladesh) I must have more details of the device Mr Lindsay wishes to communicate with, be it a teleprinter, printer, or another computer. This information must be provided by the person at the other end of the communication link.

If Mr Lindsay proposes to use the public telephone system as his communication link he will either use (a) an accoustic coupler or (b) a modem. In (a) the telephone receiver can be held close to the television loud speaker and a suitable program (in machine code) will send the appropriate noises. In (b) Mr Lindsay will have to purchase a modem from a computer dealer and have it approved and installed by the Post Office. Could Mr Lindsay please supply me with details of such a modem even though he will not wish to go through the expense of purchasing one yet. Presumable Mr Lindsay has not got the SF7000 Super Control Station which is fitted with both a parallel printer output port and also a RS232 output. I am pretty sure that the serial printer socket on the Sega can be used as a RS232 outlet, but need more details of the device at the other end, even if it is another Sega.

Finally, Mr Lindsay must tell me of the type of information he wishes to send, ie. programs or data, and how this is arranged, eg. 128 byte character strings, etc. Letters forwarded to the listed P.O. Box No. will be forwarded to me in Bangladesh by courier, usually on Fridays

PS. It is my intention to prepare programs and instructions in publishable form, but need to try them out on one or two readers first.

R.E. Templer,
c/- Worley Consultants Ltd
P.O. Box 4241,
Auckland

SEGA BASIC REVEALED

Control codes

Key operation	PRINT CHR $ (Value) ;	Function
CTRL + A	PRINT CHR $ (1) ;	NULL No character
C	–	BREAK Stops program execution.
E	5	Clears characters after the cursor.
G	7	BELL Makes beep sound.
H	8	DEL Deletes a character.
I	9	HT Horizontal tab
J	10	LF Line feed
K	11	HM Returns the cursor to the home position.
L	12	CLR Clears the screen.
M	13	CR Carriage return.
N	14	Keyboard shift (kana ↔ alphanumeric)
O	15	(* Screen shift (text screen ↔ graphic screen)
P	16	Standard character size
O	17	Character size doubled horizontally (SCREEN 2)
R	18	INS Insert
S	19	Key entry for capital letters (A-Z), no shift
T	20	Key entry for small letters (a-z), no shift
U	21	Clears the current line and returns the cursor to the left margin.
V	22	Normal mode
W	23	GRAPH Shift (key entry graph mode ↔ letter mode)
X	24	Click sound setting (on ↔ off)
Z	26	Printer selection (#1 ↔ #2)
–	28	⇨ Cursor movement
–	29	⇦ Cursor movement
–	30	⇧ Cursor movement
–	31	⇩ Cursor movement

To specify a control code in the program, enter the associated PRINT CHR$ (value)

CHAPTER 3 DISPLAY SCREEN

1. Screen

SC-3000 provides two independent screens which cannot be used simultaneously. Select the proper screen according to the operation.

Screen 1 Text screen

BASIC initially displays the text screen. Characters can be directly input to this screen. This screen cannot display the execution results of graphic statements such as LINE and CIRCLE statements.

Screen 2 Graphic screen

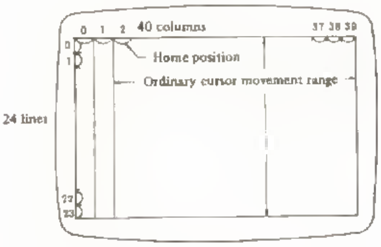
Usually, a test screen is displayed and a graphic screen is not visible. Specify a SCREEN statement to display the graphic screen. This screen displays graphics specified with graphic statements. After execution on the graphic screen is over, the program automatically displays the text screen. To continuously display the graphic screen, set an endless loop in the last line.

2.1 Text screen configuration

A text screen has a configuration of 38 columns x 24 lines. Use of the two leftmost columns on the screen are restricted, because some CRTs do not clearly display this area. Characters can be displayed in this area with a VPOKE statement. The address of the leftmost column on the screen is as follows:

&H3C00 (hexadecimal)
15360 (decimal)

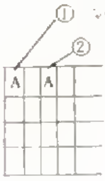
Display screen



2.2 Coordinates on text screen

① VPOKE &H3C00,&H41 or VPOKE 15360,65

② Cursor origin ($x = 0$, $y = 0$) CURSOR 0,0:PRINT "A"



Coordinates range of CURSOR statement $x = 0 \sim 37$

$y = 0 \sim 23$

The screen origin 1 is at the second column leftward from the BASIC home position 2 .
The BASIC home position implies the cursor origin, its coordinates to be specified in a CURSOR statement are x=0 and y=0
The actual screen origin is at the second column leftward from the home position, its VRAM address is &H3C00 (15360 in decimal notation).
Execute the following statements to check the above

```
① VPOKE &H3C00,&H41 or VPOKE 15360,65
```

&H41 (65 in decimal notation) is the character code of letter A

```
② CURSOR 0,0:PRINT "A"
```

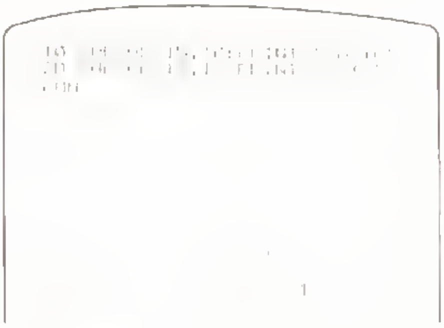
Usually, BASIC uses the home position as its origin. To display characters on the full screen, use a VPOKE statement.
When a PRINT statement is executed on a text screen, characters are displayed from the screen top. To display characters at the desired positions, move the cursor with a CURSOR statement. Characters are displayed from the position specified in the CURSOR statement. Determine the position according to the number of characters

The address calculations on the text screen are carried out as follows

$$\text{Address (text)} = y * 40 + x + \&H3C00$$
$$\text{where } (x = 0 \sim 39,\ y = 0 \sim 23)$$

For the data to be sent, the ASCII code of the corresponding character is applicable (0 ~ 255 in decimal numerals and 0 ~ &HFF in hexadecimal numerals)

Example 1.



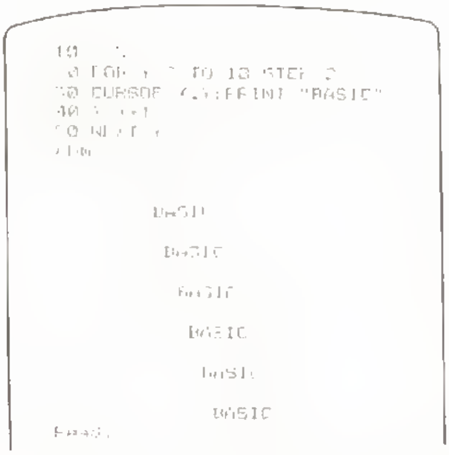
```
10 [illegible]
20 [illegible]
RUN



                    1
```

To specify the position in a CURSOR statement, enter the coordinates in the order of x- and y-axes.

Example 2



```
10 Y=1
20 FOR X=2 TO 12 STEP 2
30 CURSOR X,Y:PRINT "BASIC"
40 Y=Y+1
50 NEXT X
RUN

          BASIC

            BASIC

              BASIC

                BASIC

                  BASIC

                    BASIC
Ready
```

Variables can be used to specify coordinates in a CURSOR statement. This method is useful when changing the display position.

Example 3.

```
10 FOR Y=0 TO 15
20 CURSOR 4,Y:PRINT "I"
30 NEXT Y
```

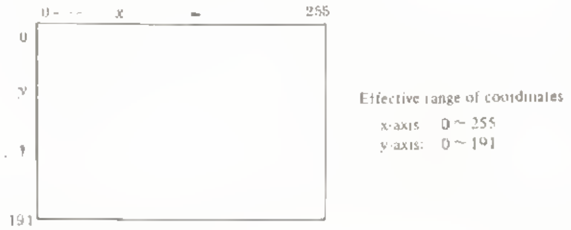
Specify a FOR statement to change only the y-coordinate.

3.1 Graphic screen configuration

A graphic screen configuration is 256 (horizontal) x 192 (vertical) picture elements. Usually, a picture element is treated as a dot In graphic screen explanations, both picture elements and dots are used, but they have the same meaning, except for special operations
The origin of a graphic screen is at the left-top corner, like for a text screen, but some CRTs may not clearly display the origin.

3.2 Coordinates on graphic screen

The graphic screen is managed in the units of dots (picture elements). In LINE, PSET and CURSOR statements, specify the coordinates in the following range



When displaying characters with a CURSOR statement, the coordinates must be specified so that the characters are not overlapped on the screen. Since a character consists of 6 (horizontal) x 8 (vertical) dots, more then 6 horizontal dots and more then 8 vertical dots must be reserved betwene the specified coordinates.

```
10 SCREEN 2,2:CLS
[illegible]
[illegible]
[illegible]
50 NEXT
[illegible] GOTO 60
RUN
Break in 60
```

Graphics screen displayed

After characters or graphics are displayed on a graphics screen, this screen is automatically changed to a text screen. To keep the graphics screen display, specify an endless loop in line 60. Press the BREAK key to terminate program execution

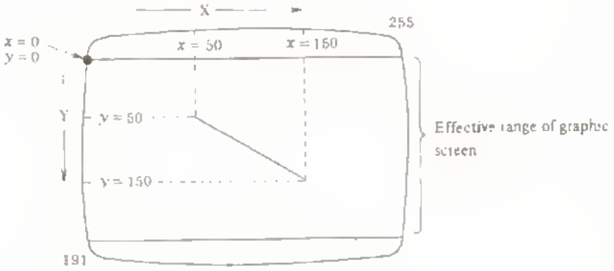
3.3 Drawing figures

Draw a line on the screen.
Before drawing a figure on the screen with a LINE or CIRCLE statement, specify the graphic screen as follows

```
SCREEN 2,2:CLS
```

Write this statement for screen selection at the beginning of the program. The program execution result can be seen on the graphic screen.
To draw a line, specify the coordinates of the line starting point and those of the line ending point in a LINE statement. The specified point must be a crossing of the x- and y-axes

```
10 SCREEN 2,2:CLS
20 LINE(50,50)-(150,150),8   ← Color number
30 GOTO 30                     (See the COLOR statement.)
RUN
Break in 30
```



For a different line, change the coordinates.
After changing the coordinates or characters with the CURSOR key, press the CR key

B specifications Drawing a box

```
LINE(50,50)-(150,150),5,B
```

A LINE statement can draw a box (7rectangular), in addition to a line. To draw a box, specify B (box). When B is specified in a LINE statement, it draws a box whose diagonal line is specified in the same LINE statement

F specification

```
LINE(50,50)-(150,150),5,BF
```

This LINE statement paints inside the box with the specified color.

Drawing a circle

To draw a circle, specify the center coordinates.

```
[illegible]
[illegible]
30 CIRCLE [illegible]
40 NEXT [illegible]
50 GOTO 50
RUN
Break in 50
```

See the explanation on the CIRCLE statement for details.

The following statements and commands are valid only on the graphic screen (They are invalid on in text screen)

```
SCREEN      POSITION
LINE        &LINE
CIRCLE      &CIRCLE
PSET        PRESET
SPRITE      MAG
```

4. Addresses on graphic screen

The PVOKE address at the leftmost end of the screen is &H0000. Data specified at this address is displayed on the screen See the explanation on a PATTERN statement for data

Example

```
VPOKE &H0011,&HFF
Ready

VPOKE &H0017,255
Ready
```

The graphic screen displayed directly with commands can be seen only for an instance To see the screen again, press the BREAK key while pressing the SHIFT key The screen is scrolled down When these keys are pressed again, the text screen is displayed.

```
10 SCREEN 2,2:CLS
20 FOR V=0 TO 7
30 READ D$
40 VPOKE &H0010+V,VAL("&H"+D$)
50 NEXT V
60 DATA 01,03,07,0F,1F,3F,7F,FF
70 CV=&H2000
80 FOR V=0 TO 7
90 READ D$
100 VPOKE &H0010+V+CV,VAL("&H"+D$)
110 NEXT V
120 DATA 5F,5F,5F,5F,8F,8F,8F,8F
130 GOTO 130
RUN
Break in 130
```

Specify data at addresses &H0010 to &H0017 to draw a triangle. The color table addresses begin at &H2000 Specify the color with by color number.

Addresses on graphic screen

Addresses on the graphic screen begins at VRAM address &H0000. An address stores 8-bit data, divided to four high-order bits and four low-order bits The high- and low-order bits are indicated in the binary notation and they are displayed in the hexadecimal notation Thus, two hexadecimal digits can be used to write the contents of an address

High-order bits ; Low-order bits	Binary notation	Hexadecimal notation	Decimal notation
	0 0 0 0 0 0 0 1	0 1	0 1
	1 0 1 0 1 0 0 0	A 8	1 6 8
	1 1 1 1 1 1 1 1	F F	2 5 5

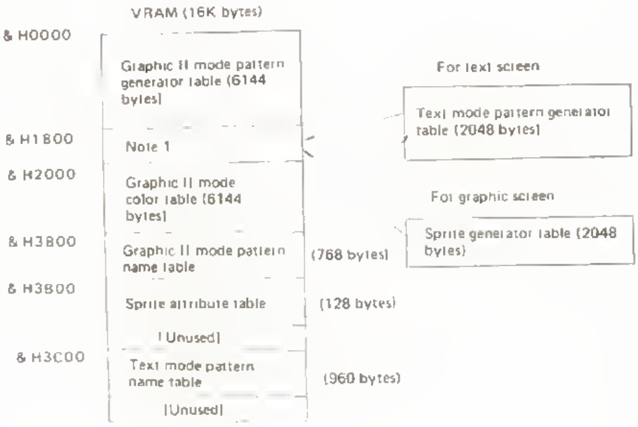
The computer can handle both decimal and hexadecimal numbers &H must be assigned to hexadecimal numbers Decimal 10 is equivalent to hexadecimal &HA.
In VRAM addressing, eight horizontal bits (one byte) has one address The bytes in the left most column are assigned addresses &H0000 to &H0007 from the top. In the same way the bytes in the next column are assigned addresses &H0008 to &H000F
A character or symbol is generated by vertically aligned eight bytes (eight addresses) (See the explanation on the PATTERN statement)
Addresses beginning with &H2000 in the color table have one-to-one correspondence with those beginning with &H0000.

```
5 REM ,     VRAM COLOR TEST
10 SCREEN 2,2:CLS
20 VPOKE&H0650,&H01
30 VPOKE&H0651,&H03
40 VPOKE&H0652,&H07
50 VPOKE&H0653,&H0F
60 VPOKE&H0654,&H1F
70 VPOKE&H0655,&H3F
80 VPOKE&H0656,&H7F
90 VPOKE&H0656,&HFF
120 VPOKE&H2650,&H5F
130 VPOKE&H2651,&H5F
140 VPOKE&H2652,&H51
150 VPOKE&H2653,&H51
160 VPOKE&H2654,&HBF
170 VPOKE&H2655,&HBF
180 VPOKE&H2656,&H6F
190 VPOKE&H2657,&HBF
200 GOTO 200
```

Write data in eight bytes having addresses &H0650 to &H0657.

The color table begins at &H2000. See the explanation on the COLOR statement for the color numbers.

VRAM MAP


VRAM (16K bytes)
& H0000
Graphic II mode pattern generator table (6144 bytes)
& H1800
Note 1
& H2000
Graphic II mode color table (6144 bytes)
& H3800
Graphic II mode pattern name table
(768 bytes)
& H3B00
Sprite attribute table
(128 bytes)
(Unused)
& H3C00
Text mode pattern name table
(960 bytes)
(Unused)
For text screen
Text mode pattern generator table (2048 bytes)
For graphic screen
Sprite generator table (2048 bytes)

Pattern generator table (text screen)

```
1000 C=255
1010 FOR A=&H1800+32*8 TO &H1800+32*8+
7:VPOKE A,C:NEXT
1020 IF INKEY$=" " THEN C=0
1030 IF INKEY$="Z" THEN C=255
1040 GOTO 1010
```

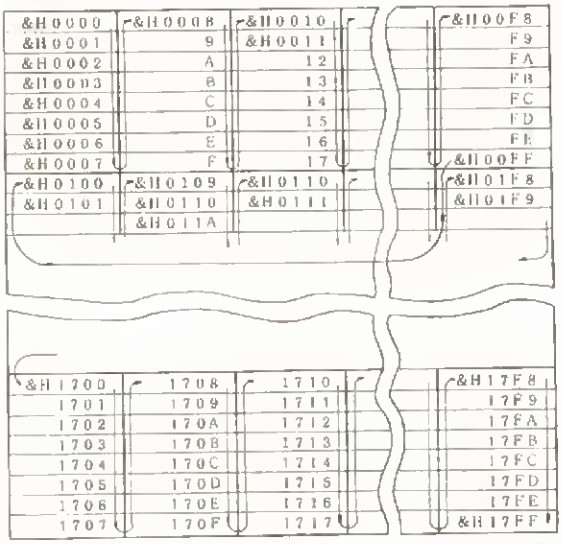
C=255 indicates a character code.

Sprite generator table (text screen)

```
2000 SCREEN 2,2:CLS
2010 FOR A=&H1800+32*8 TO &H1800+32*8+
7:VPOKE A,255:NEXT
2020 FOR M=0 TO 3
2030 MAGM
2040 FOR W=0 TO 100 : NEXT W
2050 SPRITE 0,(100,100),32,8
2060 NEXT M
2070 GOTO 2000
```

Screen top-left corner (starting address)

Screen top-right corner

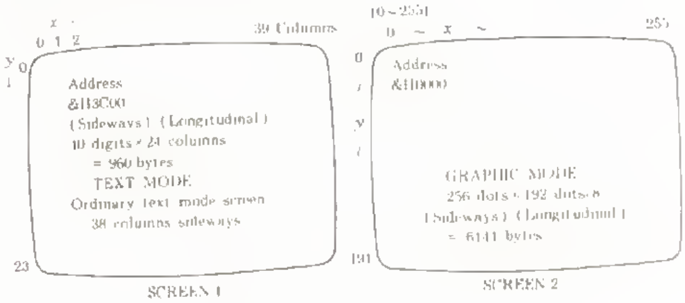


&H0000	&H0008	&H0010	&H00F8
&H0001	9	&H0011	F9
&H0002	A	12	FA
&H0003	B	13	FB
&H0004	C	14	FC
&H0005	D	15	FD
&H0006	E	16	FE
&H0007	F	17	&H00FF
&H0100	&H0109	&H0110	&H01F8
&H0101	&H0110	&H0111	&H01F9
	&H011A		
&H1700	1708	1710	&H17F8
1701	1709	1711	17F9
1702	170A	1712	17FA
1703	170B	1713	17FB
1704	170C	1714	17FC
1705	170D	1715	17FD
1706	170E	1716	17FE
1707	170F	1717	&H17FF

Screen bottom-left corner

Screen bottom-right corner (ending address)

VPOKE ADDRESS ASCII DATA (Text mode)



Part of VRAM MAP

The address calculations on the text screen are carried out as follows.

Address (text) = y * 40 + x + &H3C00
where (x = 0~39, y = 0~23)

For the data to be sent, the ASCII code of the corresponding character is applicable (32~255 in decimal numerals and &H20~&HFF in hexadecimal numerals)

Note As shown in the left figure above, the horizontal axis is deviated by 2 columns as compared to the ordinary text screen. Thus, the display position defined by CURSOR statement deviates from that defined by VPOKE, by about 2 locations in the horizontal direction

VPOKE ADDRESS, DATA (Graphic mode)

Graphic address calculations are carried out as follows

Graphic address = INT (y/8) * 256 + INT (x/8) * 8 + y MOD 8
where (y is 0 ~ 191, x is 0 ~ 255)

The address derived from the above calculations is the beginning address of 8 bits (dots) in the assigned horizontal direction. The assigned address is the x−INT (x/8) bit location counting from the left of the beginning address

The data to be sent are hexadecimal or decimal numerals displayed by the bit pattern in a horizontal row

Example

⇨ &H93 (147)

Similarly, the color table address for graphic color assignment is derived from the addition of &H2000 to the above address. The data to be sent are natural numbers (0 ~ 255) of 1B (1 Byte). The upper 4 bits of these numbers converted into binary data are the assigned color number and the lower 4 bits, the background color number. (The addresses of the graphic pattern generator table and color table respectively corresponds at 1 : 1)

Graphic color table address

= INT (y/8) * 256 + INT (x/8) * 8
+ y MOD 8 + &H2000

Where y is 0 ~ 191
x is 0 ~ 255

Color data = Assigned color No * 16 + background color No
(0 ~ 15) (0 ~ 15)

VPEEK

Use VPEEK with reference to VPOKE address. Program to read the content of the pattern generator table in VRAM

Example

```
10 AD=&H1800+&H31*8      : REM The beginning address
20 FOR A=AD TO AD+7            of REM "1" pattern
30 DA=VPEEK(A)
40 PRINT HEX$(DA
50 NEXT A
```

20			1					
60		1	1					
20			1					
20			1					
20			1					
20			1					
70		1	1	1				
00								

CHAPTER 5 COMMANDS, STATEMENTS, AND FUNCTIONS

This chapter explains the disk BASIC commands, statements, and functions in the alphabetical order. To know the classification, see the list at the end of this manual.
In the explanation, an expression indicates a function or variable expression, which may be different from arithmetic expressions. The explanation on each command, statement, or function consists of Function, Format, Description, Notes, and See Also.

Function Explains the function of the command, statement, or function.

Format: Gives a sample format. The parameters and variables are detailed in the description. Also see sample programs for actual coding.

Example. PRINT A → PRINT variable-name

In the format, complicated symbols are not used for easy understanding. You can input the format as shown.

Description Describes the function in details. Also read the explanation on the associated commands, statements, and functions, if any.
Note that, for programming, a comma (,) differs from a period (.) and a colon (:) differs from a semicolon (;).
A minus sign (-) also differs from a hyphen and longvowel sign in Kanas.

Example Understand how commands and statements run, using this example and sample programs.

Note(s) Gives programming notes.

See Also. Some commands are used in a combination. Such commands and associated commands are given here.

Error messages are listed in the Appendix. When an error message is displayed, see this list to locate the error.
The sample programs given in this manual can be applied to your own programming by modification and linking.

Command CONT (continue)

Function Resumes execution of programs previously interrupted by the BREAK key or by the STOP statement.

Format CONT

Description. A running program can be interrupted either by a STOP statement placed in the program or by hitting the BREAK key to see, for example, the content of a variable. To see the content of a variable, type the direct command PRINT "variable name" followed by CR.
Type CONT followed by CR to resume execution.
Note that an interrupted program cannot be resumed its execution if you modify or add some new lines to it during the interrupt. In such cases, the message

Can't continue error

will appear on the screen.

Example

```
LIST

1Ø FOR I=1 TO 9
2Ø FOR J=1 TO 9
3Ø PRINT I*J;
4Ø NEXT J:PRINT
5Ø STOP
6Ø NEXT I

RUN
 1 2 3 4 5 6 7 8 9
Break in 5Ø
PRINT I,J
 1                     1Ø
Ready
CONT
 2 4 6 8 1Ø 12 14 16 18
Break in 5Ø
```

Command DELETE

Function Deletes parts of a program from memory.

Format DELETE start line number – end line number
deletes the lines between the indicated line numbers inclusively

DELETE – line number	Deletes lines from the youngest up to the one indicated by the line number
DELETE line number –	Delete lines from the one indicated by the line number up to the oldest
DELETE line number	Delete only the line indicated by the line number

Description Use this command to delete a group of lines at the same time, though there are some other methods to delete parts of a program described below.

- Type the line number of the line you want to delete and hit the CR key.
- Place the cursor just after the line number of the line you want to delete, keep pressing the space key until the line except the number is erased from the screen, then hit the CR key.

Although you can erase lines from the screen with the space key or with the INS/DEL key, they remain in memory until you hit the CR key. The LIST command will display the lines you have erased from the screen without hitting the CR key.

Example:

```
DELETE 18Ø-22Ø

DELETE -25Ø

DELETE 6ØØ-

DELETE 1ØØ
```

Command LIST

Function Displays on the screen contents of the program in memory either partly or entirely.

Format LLIST
where the '– (minus sign) can be replaced by a ',' (comma)

LLIST	Display the entire program
LLIST line number	Display only the line indicated by the line number
LLIST line number – line number	Display the lines between the indicated line numbers inclusively
LLIST line number –	Display from the line indicated by the line number to the end of the program
LLIST – line number	Display from the start of the program up to the line indicated by the line number

Description Use this command to look at or modify the program in memory.
Big programs will scroll off the screen while you watch.
Hit the space key to interrupt the flow, and hit it once more to continue the flow.
Hit the BREAK key if you want to abandon the display.
You can modify the content of your program thus displayed (screen edit).

Example

```
LIST

1Ø CLS
2Ø FOR N=1 TO 2Ø
3Ø FOR M=1 TO N
4Ø PRINT "*";
5Ø NEXT M
6Ø PRINT
7Ø NEXT N
```

Command LLIST

Function. Outputs to the printer contents of the program in memory either partly or entirely.

Format LLIST

LLIST	Print the entire program
LLIST line number	Print only the line indicated by the line number
LLIST line number – line number	Print the lines between the indicated line numbers inclusively
LLIST line number –	Print from the line indicated by the line number to the end of the program
LLIST – line number	Print from the start of the program up to the line indicated by the line number

Description. Use this statement to printout parts or the entire contents of your programs, for checking or for preservance.

Example

```
LLIST

LLIST 100

LLIST -100

LLIST 100-

LLIST 100-200
```

Command LOAD

Function Loads programs from disk

Format LOAD "filename"

Description Display the names of programs on disk with the FILES command. Then move the cursor to the name of the program you want to load. Type LOAD and hit the [CR] key
The program you indicated will be loaded into memory

Example:

```
FILES

     "SAMPLE 1.bas"
     "SAMPLE 2.bas"
     "SAMPLE 3.bas"
LOAD "DEMO   1.bas"
     "DEMO   3.bas"
     "DEMO   2.bas"
     "DEMO   4.bas"
     "SOUND  1.TST"
     "SOUND  2.TST"

491 Bytes free
```

Command MERGE

Function: Merges (joins) a program on disk with the program in memory

Format: MERGE "filename"

Description: This command merges a program on disk with that in memory and thereby creates one single program in memory
Note that line numbers used in one program must not appear in the other Use RENUM for this purpose.
For example, if the program in memory has line numbers 10 thru 500, number the program on disk to be merged starting from 510.
If a same line number appeared in both programs, the contents of the line under that line number of the merging program would override the other.

Example

```
FILES

      "SAMPLE 1.bas"
      "SAMPLE 2.bas"
      "SAMPLE 3.bas"
      "DEMO   1.bas"
      "DEMO   2.bas"
      "DEMO   3.bas"
      "DEMO   4.bas"
      "SOUND  1.TST"
MERGE"SOUND  2.TST"

491 Bytes free
```

Command NEW

Function Deletes programs and resets variables in memory

Format NEW

Description: If you input lines of a program while there is some other program still in memory, they get mingled up and may lead to some unexpected result or error.
You must execute this command to delete some previous program from memory whenever you input a new program.
To see whether there is some program still in memory, use the LIST command
If there is one, delete it with this command

Example:

```
NEW
```

Command NEWON

Function: Sets the start address for the BASIC program area

Format: NEWON start address

Description. This command allocates areas of memory starting from the given address to BASIC programs, arrays, variables and so on.

Note You cannot set the address within the area for the BASIC interpreter, work area, nor in the area higher than the address previously set by the LIMIT statement.
This command, like the NEW statement, deletes state programs currently in memory.

Example:

```
NEWON &HC000
```

Command RENUM (venumber)

Function: Re-sequences the line numbers of a program

Format RENUM new line number, current line number, increment

Description. RENUM followed by the [CR] key will re-sequence the line numbers starting from 10 with an increment of 10 The line numbers appearing in a GOTO statement or in a GOSUB statement will be adjusted accordingly.
If you omit increment, it defaults to 10

Note If there is a line number that does not exist in the program and yet is referenced in one of GOTO, GOSUB, IF-THEN and RESTORE statements, then this command will cause the Undef'd line number error.

Example:

```
RENUM

RENUM 100

RENUM 300,200

RENUM 300,200,50
```

Command RUN

Function: Starts execution of a program

Format

RUN	Starts execution from the beginning of the program in memory
RUN line number	Starts execution from the line specified by line number
RUN filename	Starts execution of the named program after loading it from disk

Description Though SC-3000 has a function key for it, this statement is still useful if you want to execute a program from a given line or execute one of two programs in memory demarcated by separate line numbers.

Example:

```
LIST

100 A=100
110 PRINT A

RUN
 100
Ready

RUN 100
 0
Ready
```

Command SAVE

Function: Saves the program in memory onto floppy disks

Format: SAVE "filename.extension"

Description: Filename is a name you give to a program to somehow remember its function.
And a program can be saved only after you have christened it.
Either a complete program or a program under development can be saved.
When you create a big program, you can temporarily leave the job by saving your intermediate result, and later resume the job by loading it back.
Filename is limited to up to 8 characters optionally followed by a ' ' (period) and a 3-character extension.
If you save a program under a name, and if there is a program under that name on disk, then the program on disk will be replaced by the newly saved one. If you make some modification to a program that was previously saved on disk, then choose the same name as that of the program. But if you want to save the modified program separately, then choose partly different name from the original one.
This is because, on disk save, the place to where programs go is selected from the given floppy disk unit.

Example:

```
SAVE "SAMPLE a.TST"
```

Set also: LORD , FILES

Command UTILITY

Function: Enters the disk utility program which in turn accepts the following commands described below

Format: UTILITY [CR]

Description: UTILITY commands:

F. Format disks
C: Copy disks
B Do a BOOT

F· Disk formatting
A new disk can be used only after you have formatted it. Type this command. Set your disk into the drive and type F followed by [CR] .
Don't do anything before the cursor appears on the screen, since interruption of this command sometimes means disk destruction.
Note also that if you format a disk with some programs still in it, the programs are deleted.

C: Sometimes, though rarely, a disk may be damaged and become useless with all its plastic coverage.
You are recommended to take copies of your important programs by this command against such disaster.

B. Do a BOOT

How to copy your disks
Press the [C] key followed by the [CR] .
Set the floppy disk you want to have a copy of (SOURCE DISK), then press the space key.
Ten tracks of data will be read into the drive.
Pressing the space key at this time will start the copy. The copy will take some time. Don't interrupt the disk unit while copying since it will destroy the copy.
If you use a disk with some programs already in it, those programs will be replaced by the newly copied ones.
Repeat the above procedure 4 times to complete the copy on one side.
The message

copy complete

will appear on the screen on completion of the copy.
Copy uses a different format than that used in save. So although the BASIC system cannot be saved onto disks, it can be copied there.
Take a backup of your BASIC system by copying it to some disk.

Set also: BOOT

Command VERIFY

Function: Compares the program saved on cassette and the program in memory

Format: VERIFY "filename"

Description: This command checks whether the program in memory has been correctly saved onto cassette.
Rewind the tape upto where you started the save.
Type
VERIFY "filename"
followed by pressing down the [CR] key.
Then push the play (LOAD) key on the tape recorder.
If no difference is found between the program in memory and the program saved on cassette, the message

Verify end

will appear on the screen.
If the message would not appear, push the reset key to break the command and restart from the save.

Example:

```
VERIFY
* Verifying start
 Found xxxx
* Verifying end
```

See also: SAVE, LOAD

Statement BCIRCLE

Function: Erases lines or circles drawn on the screen.

Format: BCIRCLE (X, Y), radius, ratio, starting point, end point, BF.

Description: The statement is used in the same way as the CIRCLE statement to erase desired area, though you cannot specify color to this statement.
The color corresponding to bit "0" is chosen to erase the area.

Example:

```
10 SCREEN 2,2:CLS
20 FOR R=5 TO 1 STEP -1
30 CIRCLE(128,90),R*10,R,1,0,1,BF
40 BCIRCLE(123,90),R*9,,1,0,1,BF
50 NEXT R
60 GOTO 60
```

See Also: CIRCLE, COLOR

Statement BEEP

Function: Generates a beep sound

Format: BEEP *n*

Description: *n* must be in the range 0 thru 2

BEEP	Beep
BEEP 0	Stop beeping caused by BEEP 1
BEEP 1	Keep beeping
BEEP 2	Generate sound like peep poop

Example.

```
10 D1M A$(12)
20 FOR N=0 TO 12
30 READ A$(N)
40 PRINT A$(N);
50 BEEP
60 NEXT N
70 DATA H,O,M,E," ",C,O,M,P,U,T,E,R
RUN
HOME COMPUTER

Ready
```

Statement BLINE

Fungion: Erases by line or rectangle.

Format:
BLINE (X1, Y1) – (X2, Y2)
BLINE (X1, Y1) – (X2, Y2), BF

Description: Colors cannot be specified to the BLINE statement. The color chosen is the color of the background at the time of execution of this statement.
The color of the background corresponds to bit "0". The BF specification will erase the rectangular area determined by (X1, Y1) and (X2, Y2).

Example

```
10 SCREEN 2,2:CLS
20 FOR R=5 TO 1 STEP -1
30 CIRCLE(128,90),R*10,R,1,0,1,8F
40 BCIRCLE(128,90),R*9,,1,0,1,BF
50 NEXT R
60 GOTO 60
```

See Also: LINE, COLOR

Statement CALL

Function: Calls machine language subroutines

Format: CALL start address

Description: Since machine language programs are placed outside the BASIC program area, you must use this statement to call a machine language subroutine.

Example :

```
10 LIMIT %HDFFF :CLS
20 FDR A=&HE000 TD &HE082
30 READ D$:D=VAL("&H"+D$)
40 PDKE A,D
50 NEXT A
60 CALL %HE000
70 CURSOR 0,2:PRINT TAB(30*RND(1));"A"
80 GOTO 60
100 DATA F3,C5,D5,E5,F5,06,16,0E
110 DATA 00,CD,3D,E0,CD,58,E0,EB
120 DATA 21,83,E0,0E,27,CD,66,E0
130 DATA 77,23,0D,20,F8,EB,7D,C6
140 DATA 28,6F.30,01,24,CD,6D,E0
150 DATA 21,83,E0,0E,27,7E,CD,7D
160 DATA E0,23,0D,20,FB,10,D0,F1
170 DATA E1,D1,C1,FB,C9,C5,D5,26
180 DATA 00,68,29,29,29,54,5D,29
190 DATA 29,19,16,00,59,19,11,00
200 DATA 3C,19,D1,C1,C9,DB,BF,C9
210 DATA F5,CD,55,E0,7D,D3,BF,7C
220 DATA E6,3F,D3,BF,F1,C9,00,00
230 DATA 00,00,DB,BE,C9,F5,CD,55
240 DATA E0,7D,D3,BF,7C,E6,3F,F6
250 DATA 40,D3,8F,F1,C9,00,00,00
260 DATA D3,BE,C9
```

See also: LIMIT

Statement CIRCLE

Function: Draws circles around given points.

Format: CIRCLE (X, Y), radius, color, ratio, starting point, end point, BF.

Description: The statement draws a circle around the given point (X, Y). The arguments to this statement are explained as follows:

Radius: The scale for this quantity is measured in pixels (dots. If the length of the diameter go beyond the maximum value allowed for the coordinate, the part coming outside the coordinate will be drawn as a straight line.

Color: Specified by the color code.

Radio: Ratio of diameter to the horizontal axis explained as follows:

Is equal to 1 The ratio is 1 to 1 and the circle drawn will be a true circle

Is less than 1 And ellipse will be drawn with its horizontal diameter greater than the vertical diameter.
The allowable number of decimal places to the left of the decimal point is restricted to 1 (0.1, 0.2, . . . 1).

Is greater than 1 An ellipse with its vertical diameter greater than the horizontal diameter. The allowable values are 1.1, 1.2, . . . up to 1.

Starting point. Just imagine a clock. The circumference of any circle is so measured that the number 0 corresponds to the position the small hand points at 3 o'clock, and hence the number increases clockwise along with the circumference up to 1 which finally comes to overlap with the starting point.
You can start drawing beginning from any point on the circle by specifying a decimal fraction between 0 and 1.
The fraction can be up to two decimal places.

End point: Can be any decimal fraction between 0 and 1 inclusive The number 1 corresponds to the position the small hand points at 3 o'clock.
Supply the values like 0.25, 0.75 instead of 1.25, 1.75 since values greater than 1 will draw circles past the starting point.

B: Specifying B alone will draw line segments connecting the center of the circle to the starting and the end points.

BF: Specifying F in addition to B will paint the region drawn by the B specification.

Omitting arguments: The arguments to the CIRCLE statement can be omitted except the coordinate of the center and the radius.
If you omit color specification, the color employed by some previous statement will be used.

If you omit ratio, it defaults to 1.

If you omit the starting point specififation, it defaults to 0.

If you omit the end point specification, it defaults to 1

You cannot specify F without specifying B.

You need type only those arguments you need if you want to omit every arguments coming after a certain argument.
If you want to omit those arguments the positions of which come between other arguments, you need supply commas to indicate you have omitted those argumenta (see example below).

Example:
CIRCLE (X, Y), 50
CIRCLE (X, Y), 50, 8, , , , BF

```
10 SCREEN 2,2:CLS
20 FOR X=50 TO 200 STEP 5
30 CIRCLE (X,90),60,1,1,0,1
40 NEXT
50 FOR X=50 TO 200 STEP 5
60 BCIRCLE (X,90),60
70 NEXT
```

Statement CLOADM

Function: Loads machine language programs from cassette

Format: CLOADM "filename", load start address

Description: This statement loads the machine language program on cassette indicated by the "filename" onto memory If you specify load start address, load will start from that address.
If not, it will start from the address as previously indicated by CSAVEM.
If you omit "filename", the first program encountered on cassette during the load will be loaded.

Note The filename must be the one you christened on save, otherwise the message "Skip" will be printed and no load will be done.

Example:

```
CLOADM
* Loading start
 Found OBJ: HEX DATA
* Loading end
Ready
```

Statement	CLS	(clear screen)

Function: Clears the currently active part of the screen

Format. CLS

Description: This statement erases everything, programs and results displayed by the previous run of some programs, from the current window (the part of the screen which is currently active). No other part of the screen other than currently active window will be affected by this statement

Example:

```
SCREEN 2,2:CLS
Ready
```

```
10 CLS
20 FOR E=0 TO 100
30 PRINT I:
40 NEXT E
```

Statement	COLOR

Function: Sets color on the screen.

Format: For the text window.
COLOR color code for character, color code for background for the graphics screen.
For the graphics screen
COLOR c1, c0, (X1, Y1) – (X2, Y2), cb.

Description: c1: The color corresponding to bit "1" (clor for characters and lines)
The color applies to
- Characters printed by the PRINT statement
- Points or lines drawn by the PSET, LINE or CIRCLE statements
- Areas to be painted by the BF specification to the LINE or BLINE statement

c0: The color corresponding to bit "0" (background color)
Which applies to
- The window and the background after execution of the CLS statement
- The part with bit "1" reset by the PRESET, BLINE or BCIRCLE statements.

(X1, Y1) – (X2, Y2)
Paint inside the rectangle having the segment connecting (X1, Y1) and (X2, Y2) as its diagonal.
The c0 argument must be specified.

cb: Color of the backdrop
(Equivalent to "transparency")
The backdrop is the upper and the lower margins of the screen in which neither characters or symbols, nor points or lines can be drawn.
"Transparency" corresponds to the color of these margins.
Each color has a code associated with it:

Color Code Table

Color code	Color	Color code	Color	Color code	Color
0	Transparency	6	Dark red	12	Dark green
1	Black	7	Light blue (cyane)	13	Mazenta
2	Green	8	Red	14	Grey
3	Light green	9	Light red	15	White
4	Dark blue	10	Dark yellow		
5	Light blue	11	Light yellow		

Note: The unit of area with which color can change from one to another consists of a horizontal row of 8 successive pixels (pixel is equivalent to picture element, which is the smallest dot of which characters or figures are comprised). Any area consisting of a successive row of 8 pixels can contain up to 2 colors including the color for the background, which means color for points or lines cannot vary within the area. And if you specify 3 different colors to paint the area, the entire area will be painted by the 3rd color specified.
Remind this fact when you use the LINE, CIRCLE or the PSET statement
The above mentioned units are not placed arbitrarily on the screen. On any one line of the screen, the first unit consists of from 0 to 7th pixels, next from 8 to 15 pixels, and so on.

Additional Information: You can find in the explanation of the graphics mode for the SC-3000 those words such as pixel, dot and bit.

A pixel is the least unit of point used to draw figures in the graphics mode
A bit is the least unit your computer can understand.
A dot is a least unit for drawing pictures under a certain condition.

In the world of the SC-3000 graphics mode, pixel, dot or bit are usually synonyms each other.

Example:

```
10 SCREEN 2,2:CLS
20 FOR A=1 TO 12
30 COLOR A,15
40 FOR I=1 TO 82:PRINTCHR$(144);:NEXT
50 NEXT A
60 FOR C=1 TO 15
70 FOR Y=0 TO 191 STEP 2
80 COLOR ,C,(0,Y)-(255,Y)
90 NEXT Y
100 FOR X=0 TO 255 STEP 3
110 COLOR ,C,(X,0)-(X,191)
120 NEXT X,C
130 GOTO 60
```

Statement	CONSOLE

Function: Sets the cursor scroll limit for the text window, controls the on/off of the click sound, switching between upper and lower cases for characters, and selects printer (#1, #2).

Format: CONSOLE u, l, c, s, p
where
u: Scroll upper limit (0 thru 22)
l. Scroll length (greater than or equal to 2)
c. Click sound on/off (0 = off, 1 = on)
s Change case (0 = upper case, 1 = lower case)
p Select printer (1 = printer #1, 2 = printer #2)
At boot time, each value is initialized as u = 0, l = 24, c = 1, s = 0, p = 1

Description: The values set by this statement are not altered (including program abort) unless delivered by the reset key or reset by another CONSOLE statement
Printer #1 corresponds to the SEGA SP-400
Printer #2 is for a Centronics type printer.

Example:

```
LIST
100 CLS:N=24
110 FOR I=1 TO 7
120 READ A$
130 GOSUB 260
140 CONSOLE I,N
150 NEXT I
160 DATA " ●●●    ●●●●   ●●●     ●  "
170 DATA "●   ●  ●      ●   ●   ● ● "
180 DATA "●      ●      ●      ●   ●"
190 DATA " ●●●   ●●●●   ● ●●●  ●   ●"
200 DATA "    ●  ●      ●   ●  ●●●●●"
210 DATA "●   ●  ●      ●   ●  ●   ●"
220 DATA " ●●●    ●●●●   ●●●●  ●   ●"
230 CONSOLE 0,24
240 CURSOR 0,10
250 END
260 CURSOR 0,23
270 FOR P=1 TO 30
280 PRINT MID$(A$,P.1);:BEEP
290 NEXT P
300 N=N-1
310 FOR J=1 TO N
320 PRINT
330 NEXT J
340 RETURN
```

Statement CSAVEM

Function. Saves machine language programs onto cassette

Format: CSAVEM "filename", start address, end address

Description: This statement saves the machine language program in memory onto cassette tapes. Filename in this case is limited to up to 16 characters and has no extension

Example:

```
CSAVEM "HEX DATA ",&HF000,&HFFFF
* Saving start
* Saving end
Ready
```

Statement CURSDR

Function: Sets the cursor on the specified position

Format: CURSOR horizontal position, vertical position

Description: When used on the text window

horizontal position must be in the range: 0 thru 37
vertical position must be in the range: 0 thru 23

When used on the graphics window

horizontal position must be in the range 0 thru 255
vertical position must be in the range 0 thru 191

In either of the above cases, ranges out of the ones as specified will cause "Statement parameter error."
If you change the origin of a coordinate on the graphics window with the POSITION statement, the range of the values which can be handled on the coordinate will also change.
The positive range (with respect to the origin) will now be bounded by maximum value – specified coordinate , while the negative range by the negative value of the origin.
The range in this case of course means integer range.

Example:

```
CURSDR 18,12 :PRINT "A"
```

```
10 SCREEN 2,2:CLS
20 CURSOR 125,95:PRINT "A"
```

See also: POSITIDN

Statement DATA

Function: Supplied data to a READ statement

Format: DATA numeric value or character string

Description: Multiple number of data can be supplied to this statement as in DATA 1, 2, 3, 4 where comma is used to distinguish each datum.
Character strings need not be enclosed in double quores except

(;), (,), (")

which must be double-quoted as shown below:

";",",",""""

If a numeric datum corresponds to a character string variable in the corresponding READ statement, the darum will be regarded as a character string and hence cannot be used in a numeric expression.
The number of data in the statement and the number of arguments in the corresponding READ statement must be the same.
If the number of data in the statement and the number of arguments in the corresponding READ statement, only the data corresponding to the arguments will be utilized. An error will occur if the number of arguments in a READ statement exceeds that of the data in the corresponding DATA statement.
The READ statement, once executed, reads data from the corresponding DATA statement independent of the latter statement's position in the program.

Example:

```
LIST

10 READ A,B,C,D
20 PRINT A+B+C+D
100 DATA 1,2,3,4

RUN
 10
Ready
```

See also: READ, RESTORE

Statement DEF FN

Function: Defines user functions

Format. DEF FN function name (argument) = function definition expression

Description: Function name must be longer than 2 characters Including the head "FN "
The third character of any function name must be alphabetic, and no reserved word (such as command names) must appear in it.

(Correct) FNA FNB FNCD
(Wrong) FNABS FN1 FMC

Function names are distinguished only by up to 2 characters following "FN "
This means two function names with the same two characrers after "FN" are indistinguishable
For example, the following two function names

FNSEGA and FNSE

are regarded to be the same.
Also, the value of the argument you supply to your function does not change after the function invocation.

Example: $\sinh x = \frac{e^x - e^{-x}}{2}$, $\cosh x = \frac{e^x + e^{-x}}{2}$

Let's define the above functions

```
10 DEF FNSH(X)=(EXP(X)-EXP(-X))/2
20 DEF FNCH(X)=(EXP(X)+EXP(-X))/2
30 INPUT "X=";X
40 PRINT "sinh(x)=";FNSH(X)
50 PRINT "cosh(x)=";FNCH(X)
```

Statement DIM

Function. Declares arrays. Dimension is limited up to 3.

Format: DIM arrayname (subscript range)
DIM A (20)
DIM B$ (5,5), DIM C (2, 3, 4)

Description. Arrays are either one-dimensional array or multi-dimensional. Multi-dimensional array is limited up to 3-dimensional array.
Declaring the one-dimensional array

A (5)

where the number in the parentheses is called a subcript, is equivalent to declaring the following six variables:

A (0), A (1), A (2), A (3), A (4), A (5)

Character string arrays can be declared also.
You can use an array element without the necessary declaration but in that case the subscript range is 10.
A two-dimensional array

B (5,5)

and a three-dimensional array

C (3,3,3)

Example:

```
LIST

10 CLS
20 DIM A(9,9)
30 FOR J=1 TO 9
40 FOR K=1 TO 9
50 A(J,K)=J*K
60 IF J*K<10 THEN PRINT" ";
70 PRINT A(J,K):
80 NEXT K
90 PRINT
100 NEXT J
```

See also: ERASE

Statement END

Function: Puts an end to programs

Format: END

Description: Append this statement to the end of a program if the flow of the program follows the line number.
But those programs having subroutines at their tail must end somewhere before the last statement Put an end to them with this statement.

Example:

```
10 GOSUB 100
20 PRINT"  LET BASIC STUDY"
30 END
100 FOR N=0 TO 37
200 PRINT"*"120 NEXT N
130 RETURN
```

Statement ERASE

Function. Cancels array declarations

Format: ERASE
ERASE arrayname, arrayname

Description: If you omit arrayname, all array declarations will be canceled.
With a program, you cannot declare arrays twice under a same name. But if the program flow forces you to do so, use this statement to cancel the former declaration.

Example:

```
100 ERASE
   •
   •
   •
200 ERASE A,B$
```

Statement FOR–NEXT–STEP

Function. Repeats lines inserted between the FOR and the NEXT statements

Format: FOR numeric variable = initial value TO final value STEP increment
NEXT numeric variable

Description. You can insert between the FOR and the NEXT statements the part of your program you want to repeat many times. When the program reaches to the NEXT statement, the variable gets incremented by the amount you specified just after STEP, and that part of yours between the FOR and the NEXT statements is repeated once more.
When the value of the variable reaches to the final value you specified just after TO, then those statement just after the NEXT statement will begin to execute.
If you omit the STEP increment part, the increment defaults to 1.
Note that the increment must be a negative value to "count down" if the initial value is greater than the final value
The FOR–NEXT statement can be nested (you can put a FOR–NEXT statement within another FOR–NEXT statement), but in which case you must use distinct variables
A convenient way is to have the NEXT statement two variables, one for the inner and the other for the outer FOR, but in that case you must put the variable for the inner FOR the first.
The depth of one nest can be up to 8.
In the following cases, statements following the FOR statement is executed only once

Initial value is smaller than final value and increment is negative
Initial value is greater than final value and increment is positive
Initial value is equal to final value
There is no NEXT statement

Statement GOSUB – RETURN

Function: Calls and executes a subroutine; after subroutine execution, returns to the line succeeding the GOSUB statement.

Format: GOSUB line-number
⁝
RETURN

Description: Line-number specifies the first line number of the subroutine. The subroutine is an independent program placed inside or at the end of the program and is called when necessary. Specify a RETURN statement specifies returning to the line succeeding the GOSUB statement.
Control can transfer from a subroutine to another subroutine in a nested subroutine structure.
Subroutinea can be nested up to level 8; if this is exceeded, a GOSUB nesting error occurs.

Note: The control returned by a RETURN statement must not go to a RETURN statement. If a RETURN statement is encountered by a statement other than a GOSUB statement, a RETURN without GOSUB error occurs.

Example:

```
10 INPUT"score";A
20 IF A>=65 THEN GOSUB 50
30 IF A<65 THEN GOSUB 70
40 GOTO 10
50 PRINT"acceptable"
60 RETURN
70 PRINT"unacceptable
80 RETURN
```

See Also: ON GOSUB

Statement GOTO

Function: Jumps to the specified line number.

Format: GOTO line-number

Description: Program execution starts from the smallest line number. When a GOTO statement is encountered, the control unconditionally jumps to the specified line number.
A direct command can specify starting program execution from an arbitrary line number specified in a GOTO statement. In this case, the variable value remains unchanged. The variable value can be known by directly executing a PRINT variable.
When a RUN or RUN line-number is executed, all variable values are cleared.

Example:

```
10 INPUT"A=";A
20 INPUT"B=";B
30 C=A+B
40 PRINT"A+B=";C
50 GOTO 10
```

See Also: ON GOTO

Statement HCOPY (hard copy)

Function: Outputs to the printer current screen image

Format: HCOPY
HCOPY *n*, enlargement

Description: This statement lets you printout current images of the text window or the graphics window.
The function of this statement is governed by the type of your printer:

The SEGA printer SP-400
Only the text window can be printed-out. Also the printable characters are restricted to the ASCII codes only and the graphics symbols for the SC-3000 cannot be printed.

The EPSON RP-80II (Centronics type)
Both the text window and the graphics window can be printed-out. Select printer mode according to the following instructions prior to the execution of HCOPY:

○ Hit the Z key while keeping down the control key
○ Supply 2 to the CONSOLE statement to select the printer

After you have switched to the printer mode #2, select the window as follows:

HCOPY 1 Printout the text window (1 can be omitted)
HCOPY 2 (Graphics window), enlargement

If you omit *n* in your program, the window currently active will be printed-out.
The enlargement is explained as follows

0: Standard (0 can be omitted)
1: Double the scale of horizontal direction
2: Double the scale of vertical direction
3: Double the scale of both directions

Example:

```
HCOPY
```

Statement IF–THEN

Function: Conditionally jumps to the specified line number or executes the statement(s) following THEN

Format: IF conditional expression THEN line number
IF conditional expression GOTO line number
IF conditional expression THEN statement(s)

Description: If the conditional expression is true, then either the statement placed after THEN, or the statement indicated by the line number supplied after GOTO or THEN is executed.
If the condition is false, the line immediately following the IF–THEN statement is executed.
Conditional expressions are usually comparisons or logical operations. A conditional expression takes the value −1 if the condition is true, and 0 otherwise.
You can place more than one statement after THEN, in which case those statements are executed only when the condition is true

Example:

```
10 INPUT"score ";A
20 IF A<50 THEN PRINT"unacceptable"
30 IF A>49 AND A<60 THEN PRINT"borderline"
40 IF A>59 AND A<70 THEN PRINT"acceptable"
50 IF A>69 THEN PRINT"light staff "
60 GOTO 10
```

Statement INPUT

Function: Gets inputs of numeric values and strings of characters from keyboard

Format: INPUT A, B$ numeric or character string variable
INPUT "prompt"; numeric or character string variable

Description: This statement, once executed in your program, waits for your input by putting a "?" (question mark) onto the screen. If you supply "prompt," then the waiting signal will be "prompt" with no question mark added to it.
Numbers or characters typed in response to the waiting signal (prompt) followed by the CR key will be assigned to the corresponding variables.
If the statement has more than one variable, the waiting signal for the second variable and on will be the string of two consecutive question marks (??).

INPUT A, B, C

Character strings need not be enclosed in double quotes.
The statement displays Redo from start and waits for your input once again if it finds type mismatch between the variable and the data you input.
If you hit just the CR key (without any other characters or numbers) to the statement's input request, following values will be assigned to the variables:

0 when the variable is numeric
null string when the variable is character string.

Null string is the string having no characters in it.

Example:

```
10 CLS
20 CURSOR 10,3:PRINT"menu"
30 CURSOR 10,6:PRINT"1...drink"
40 CURSOR 10,8:PRINT"2...food "
50 CURSOR 10,10:PRINT"3...dessert"
60 CURSOR 10,13:INPUT "order ?";A
70 ON A GOSUB 100,200,300
80 GOTO 60
100 CURSOR 10,16:PRINT"                "
110 CURSOR 10,16:PRINT"coffee...$1.00"
120 RETURN
200 CURSOR 10,16:PRINT"                "
210 CURSOR 10,16:PRINT"cake... $2.00"
220 RETURN
300 CURSOR 10.16:PRINT"                "
310 CURSOR 10.16:PRINT"melon... $300"
320 RETURN
```

Statement LET

Function: Stores (assigns) the right-hand-side value to the left-hand-side variable or an array element

Format: LET variable or an array element = numeric expression
LET character string variable or character string array element = character string

Description: LET is an assignment statement storing the right-hand-side value to the left-hand-side variable or array element
Typing, without "LET"

X = 5

has quite the same effect as typing

LET X = 5

The equal sign '=' above does not mean, as does in mathematics, the equality between the right-hand-side and the left-hand-side.

Example:

```
LIST

10 LET A=3
20 LET B=5
30 LET C=A+B
40 PRINT C
50 END

RUN
 8
Ready
```

Statement LIMIT

Function: Sets the end address for the BASIC program area

Format: LIMIT end address

Description: This statement sets the limit for the BASIC program area, and thereby sets the limit for the user workable area.
You cannot specify an address within the work area for the BASIC interpreter, nor lower than the address as previously set by the NEWON statement.
After the execution of this statement, you can use freely the area higher than or equal to the specified address. The BASIC interpreter will not touch this area.

Example:

```
LIMIT &HFFFF
```

Statement LINE

Function: Draws line segment connecting specified coordinates.

Format: LINE (X1, Y1) – (X2, Y2) , color code
where
X = horizontal coordinate in the range 0 thru 255
Y = vertical coordinate in the range 0 thru 191

Description: This statement draws the line segment starting from (X1, Y1) and ending at (X2, Y2).
If the origin of the coordinate has been moved by the coordinate to appear, the horizontal distance as well as the vertical distance of these two points must not exceed the range specified above.

Additional function B: Draws a rectangle.

LINE (X1, Y1) – (X2, Y2), color code, B

The above statent draws the rectangle having the line segment connecting (X1, Y1) and (X2, Y2) as its diagonal. What is more, you can paint inside the rectangle by saying:

LINE (X1, Y1) – (X2, Y2), color code, BF

where the color is specified by the color code.
If you omit the starting coordinate (X1, Y1), the draw will begin from the latest point utilized not only by the LINE statement, but the BLILE, PSET or the PRESET statements.

Example :

```
10 SCREEN 2,2:CLS
20 LINE(50,50)-(150,50),1
30 LINE-(50.150),8

10 SCREEN 2,2:CLS
20 FOR C=0 TO 15
30 LINE(90,50)-(160,100),C,B
40 FOR A=0 TO 300:NEXT A
50 NEXT C
60 GOTO 60
```

See Also: COLOR

Statement LPRINT

Function. Output to the printer values or character strings

Format.

LPRINT	A or A$	Numeric variable or character string variable
LPRINT	A$;B,C	
LPRINT	"X"	Character string
L? A		The "PRINT" can be replaced with "?"

Description. This statement is the same with the PRINT statement except the result is written to the printer.
"LPRINT" can be abbreviated to "L?".

Note Refer to the manual for your printer before using this statement since there can be a variety of specifications among various printers or from interface to interface.

See also PRINT

Statement MAG (magnitude)

Function: Sets size and magnitude of sprites

Format MAG numeric value

Description: Various sizes of sprites can be set by supplying to the MAG statement integers in the range 0 thru 3

MAG 0: Draws 8 by 8-dots' figures in the frame of 8 by 8 picture elements

MAG 1 Draws 16 by 16 dots' figures in the frame of 16 by 16 picture elements by combining 4 patterns of 8 by 8 picture elements (S#0–S#3, S#4–S#8, . . . , S#253–S#255)

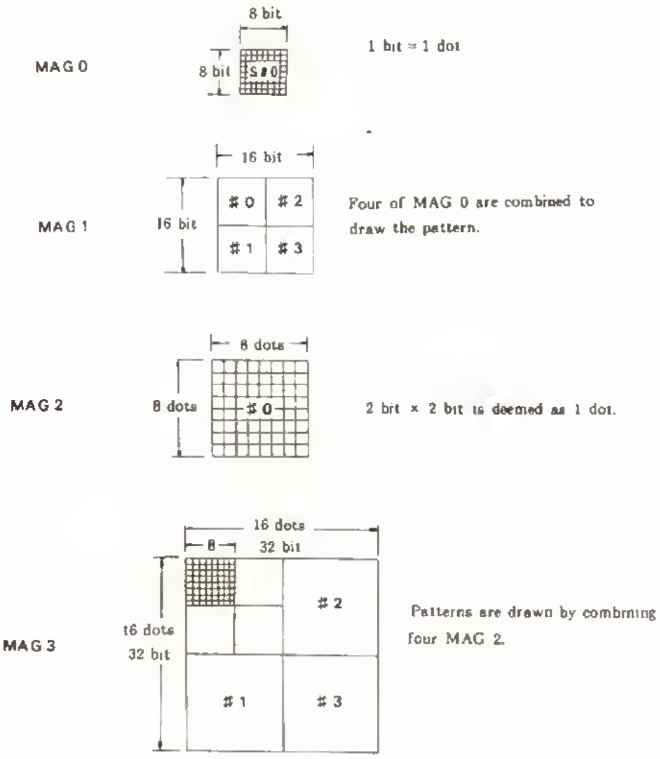
MAG 2: Double the size of the pictures drawn by MAG 0. 8 by 8 dots' figures will be drawn in the frame of 16 by 16 picture elements, 1 dot becoming equivalent to 2 by 2 picture elements.

MAG 3: Double the size of figures drawn by MAG 1. 16 by 16-dots' figures will be drawn in the frame of 32 by 32 picture elements by combining 4 patterns of 16 by 16 picture elements. 2 by 2 picture element becomes equivalent to 1 dot.

Combining 4 patterns to create a figure as in the cases MAG 1 and MAG 3 above can be done with a single SPRITE statement. Since sprite names are synonyms for pattern numbers (S#number), you can, for example, let one pattern number among the group S#0–S#3 be a sprite name to automatically construct the S#0–S#3 pattern.

Note In cases MAG 1 and MAG 3 above, the possible combinations of patterns are not arbitrary.
If you make some mistake in numbering the patterns, the resulting figures will be different from what you expect.

Figure
The MAG statement is used to specify the scale of figures drawn by the PATTERN statement.
In this figure, one picture element corresponds to one bit.


MAG 0
8 bit
8 bit
S#0
1 bit = 1 dot
MAG 1
16 bit
16 bit
#0
#2
#1
#3
Four of MAG 0 are combined to draw the pattern.
MAG 2
8 dots
8 dots
#0
2 bit × 2 bit is deemed as 1 dot.
MAG 3
16 dots
32 bit
8
16 dots
32 bit
#2
#1
#3
Patterns are drawn by combining four MAG 2.

Example :

```
10 REM  --- MAG & PATTERN TEST ---
20 SCREEN 2,2:CLS
30 PATTERN S#0,"0103070F1F3F7FFF"
40 PATTERN S#1,"FF00FF00FF00FF00"
50 PATTERN S#2,"80C0E0F0F8FCFEFF"
60 PATTERN S#3,"AAAAAAAAAAAAAAAA"
70 X=32:Y=90:XX=0
80 PRINTCHR$(17)
90 MAG M
100 FOR T=0 TO 3
110 BLINE(0,16)-(255,24),,BF
120 CURSOR 0,0:PRINT"  MAG & PATTERN T
EST"
130 CURSOR 0,16:PRINT"  MAG";M;":PATTE
RN S#";:CURSOR204,16:PRINT T
140 SPRITE 2,(X,Y),T,T+1
150 SPRITE 0,(X+32,Y),T,T+3
160 SPRITE 3,(X+64,Y),T,T+5
170 SPRITE 5,(X+96,Y),T,T+7
180 FOR W=0 TO 130
190 SPRITE 1,(160,W),T,14
200 NEXT W
210 FOR WT=0 TO 100:NEXT WT
220 NEXT T
230 M=M+1:IF M=4 THEN M=0:T=0
240 GOTO 90
```

Statement ON GOSUB

Function: Jumps ro one of the subroutines specified by the line numbers according to the variable

Format. ON variable GOSUB line number, line number, line number

Description: Jumps to one of the subroutines indicated by the line numbers specified after GOTO according to the value of the variable previously assigned by a numeric expression or by an INPUT statement.
The value is an integer and must be taken in the range 1 thru the number of line numbers you specify after GOTO, each integer corresponding to each line number.
The RETURN statement is used on return from subroutines

Example:

```
10 CLS
20 CURSOR 10,3:PRINT"menu"
30 CURSOR 10,6:PRINT"1...drink"
40 CURSOR 10,8:PRINT"2...food"
50 CURSOR 10,10:PRINT"3...dessert"
60 CURSOR 10,13:INPUT"order?";A
70 ON A GOSUB 100,200,300
80 GOTO 60
100 CURSOR 10,16:PRINT"               "
110 CURSOR 10,16:PRINT"coffee...¥300"
120 RETURN
200 CURSOR 10,16:PRINT"               "
210 CURSOR 10,16:PRINT"cake... ¥200"
220 RETURN
300 CURSOR 10,16:PRINT"               "
310 CURSOR 10,16:PRINT"melon... ¥250"
320 RETURN
```

Statement ON GOTO

Function: Jumps to one of the specified lines according to the variable

Format ON variable GOTO line number, line number, line number

Description: Jumps to one of the lines specified after GOTO according to the value of the variable previously assigned by a numeric expression or by an INPUT statement.
The value is an integer and must be taken in the range 1 thru the number of line numbers you specify after GOTO, each integer corresponding to each line number.
If the value got greater than the number of line numbers, the line immediately following this starement would be executed

Example:

```
10 INPUT"order";A
20 ON A GOTO 100,200,300
30 GOTO 10
100 PRINT "coffee":GOTO 10
200 PRINT "cake":GOTO 10
300 PRINT "milk":GOTO 10

RUN
order ?1
coffee
order ?2
cake
order ?3
milk
order ?
Break in 10
```

Statement OUT

Function Outputs data to specified output port.

Format. OUT output port number, data

Description: Output port number are predetermined by the system for outputting data to external devices.

Example:

```
10 SOUND 1.262,0
20 SOUND 2,294.0
30 SOUND 2.330,0
40 FOR A=0 TO 15 STEP .5
50 OUT &H7F.&H90+A:REM
50 OUT &H7F,&HB0+A:REM  turn of tone
70 OUT &H7F.&HD0+A:REM
80 NEXT A
90 GOTO 40
```

Statement PAINT

Function. Paints inside or outside areas formed by bits 1.

Format: PAINT (X, Y), color code

Description: Use this statement to paint inside or outside those areas drawn by the LINE or the CIRCLE statement. But note that even a one-bit hole in such regions will cause the color wooze out from the hole.
Make sure lines have no break points on them.
Use the RESET key to interrupt or stop the statement since the painting cannot be interrupted by the BREAK key.

Example:

```
10 SCREEN 2,2:CLS
20 FOR I=0 TO 255 STEP 16
30 LINE(I,0)-(I,191):NEXT I
40 FOR I=0 TO 191 STEP 16
50 LINE(0,I)-(255,I):NEXT I
60 C=RND(1)*16
70 X=RND(1)*256:Y=RND(1)*192
80 PAINT(X,Y),C
90 GOTO 60
```

Statement PATTERN

Function: Sets character or sprite pattern.

Format. To set a character partem
PATTERN C# character code, numeric character string where character code must be in the range 32 thru 255 to set a sprite pattern.

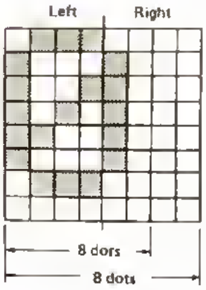
PATTERN S# sprite name, numeric character string

Where sprite name is an integer in the range 0 thru 255 which can also be supplied as a hexadecimal number.

Description. In both of the above formats, the numeric character string must be supplied as a hexadecimal number.

The format for character patterns differs from that of sprite patterns:

Character pattern (characters and symbols that can be input from the keyboard).
The pattern is constructed out of the 8-by-8 dots' square (see figure below)
In this frame, the bottom row and the rightmost column are left blank so that characters do not touch each other vertically and horizontally.
Besides, the rightmost two columns are ignored for character patterns.
So only the first 6 columns and the first 7 rows in the frame are utilized for character patterns.



Binary representation		Hexadecimal representation	
Left	Right	Left	Right
0111	0000	7	0
1000	1000	8	8
1001	1000	9	8
1010	1000	A	8
1100	1000	C	8
1000	1000	8	8
0111	0000	7	0
0000	0000	0	0

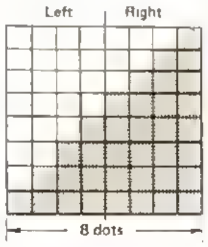
Shadowed square = bit 1   Blank square = bit 0

```
PATTERN C#92, "708898A8C8887000" [CR]
```

Now type

92 in the C#92 above corresponds to the character "¥" the ascii code of which is 92.
Now press the "¥" key, and you will see a "0" appear on the screen.
This means the pattern corresponding to the ascii code 92 has just been replaced by the one you input with the PATTERN statement.
Since patterns defined in this way remain unchanged until you power-off the computer or re-boot the system, you must be careful not to meddle the ordinary keys with your patterns.
If you do that, talking to your computer such as inputting programs will become much confusing

Sprite pattern (used only on the graphics window)
Like character patterns, sprite patterns are constructed out of 8-by-8 dots' square. But unlike them, you can use the entire square for the sprite patterns.



0000	0001	0	1
0000	0011	0	3
0000	0111	0	7
0000	1111	0	F
0001	1111	1	F
0011	1111	3	F
0111	1111	7	F
1111	1111	F	F

```
PATTERN S#0, "0103070F1F3F7FFF"
```

Note: The PATTERN statement uses different formats for character patterns and for sprite patterns:

C# for character patterns
S# for sprite patterns

See Also: SPRITE, MAG

Designing a pattern

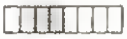
Get a sheet of graph section paper and draw an 8-by-8 square on it.
Now shadow appropriate squares in the frame to realize your image of the pattern you want.
Write sequences of 0's and 1's beside each row in the frame following the rule.

a shadowed square corresponds to 1

a blank square corresponds to 0

In this way you get 8 rows of binary numbers, each binary number corresponding to each row in the frame.
Now divide each binary number in two from the center to get two binary numbers having 4 places.
You have now 8 rows of 2 binary numbers
Translate them into hexadecimal using the conversion table given below.

For example, the row



becomes

0 1 1 1 0 0 0 0

the left half of which is 7 in hexadecimal and the right half 0 yeilding "70".
Supply the hexadecimal number thus got to the PATTERN statement and you will see, by pressing an appropriate key, your pattern displayed on the screen.
Use 8 by 8 square for graphics patterns, and 8 by 6 sequare for character patterns.

Conversion Table

10 decimal	2 binary	16 hexadecimal
0	0000	0
1	0001	1
2	0010 Shift	2
3	0011	3
3	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10 Shift	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10 Shift

```
10 PATTERN C#48,"3048484848483000"
20 PATTERN C#49,"2060202020207000"
30 PATTERN C#50,"708888102040F800"
40 A$(0)="20A8702070A82000"
50 A$(1)="0000000000000000"
60 A$(2)=A$(1)
70 CLS
80 FOR B=9 TO 3 STEP -3
90 X1=19-B*2:X2=19+B*2
100 Y1=11-B:Y2=11+B
110 C=0:FOR L=X1 TO X2
120 Y=Y1:X=L:GOSUB 260
130 Y=Y2:X=X2-L+X1:GOSUB 260
140 C=C+1:NEXT L
150 C=0:FOR L=Y1 TO Y2
160 X=X1:Y=Y2-L+Y1:GOSUB 260
170 X=X2:Y=L:GOSUB 260
180 C=C+1:NEXT L
190 NEXT B
200 FOR K=0 TO 3:C=0:FOR L=0 TO 3
210 PATTERN C#48,A$(CMOD3)
220 PATTERN C#49,A$((C+2)MOD3)
230 PATTERN C#50,A$((C+1)MOD3)
240 C=C+1:NEXT L,K
250 GOTO 200
260 VPOKE &H3C00+X+Y*40,CMOD3+48
270 RETURN
```

Statement PSET

Function: Puts dot on the specified coordinate

Format: PSET (X, Y), color code

Description: The statement puts dot (a pixel) on the coordinate (X, Y) on the screen.
If you omit color code, the color used by a previous statement will be used.

Example:

```
10 SCREEN 2,2:CLS
20 X=0:Y=95:E=1
30 PSET(X,Y),8
40 X=X+1:Y=Y+E
50 IF Y=120 THEN E=-1
60 IF Y=85 THEN E=1
70 IF X=250 THEN END
80 GOTO 30
```

See Also: PRESET, COLOR

Statement READ

Function Reads data specified by a DATA statement

Format READ variable name or array name
READ A or READ A, B, C$

Description This statement must be paired with a DATA statement.
The READ statement reads the data supplied to a DATA statement placed anywhere in the program.
The variables to a READ statement can either be numeric or string, but if the type of a datum to be read differs from that implied by the variable, the mismatch error will occur
The READ statement can take a multiple number of arguments as in

READ A, A$, B, B$

but the number of arguments in the statement must agree with the number of data in the corresponding DATA statement:

```
READ  A,  A$,  B,  B$
      ↓   ↓    ↓   ↓          types of variables
DATA 10, apple, 5, orange
```

If the number of arguments to a READ statement exceeds that of the data in the corresponding DATA statement, an error will occur. If, on the contrary, the number of data in a DATA statement exceeds that of arguments in the corresponding READ statement, the remaining data will either be ignored or read by the next READ statement.
In case there are more than one DATA statement in a program, the READ statement is used to read them all.

Example:

```
LIST

10 READ A,B,C,D
20 PRINT A+B+C+D
100 DATA 1,2,3,4

RUN
 10
Ready
```

See also DATA, RESTORE

Command REM (remarks)

Function: Marks comment

Format: REM

Description Use this statement to insert remarks in your program.
The BAS1C interpreter will ignore the lines beginning with REM

Example:

```
10 REM*** CALCULATOR ***
20 CLS
30 PRINT 2+3
```

Statement RESTORE

Function. Specifies a DATA statement to be read by the next READ statement

Format: RESTORE line number

Description: In a program with more than one DATA statement this statement is used to declare that the DATA statement associated with the given line number is to be read next.
If you omit line number, the next instance of a READ statement will read from the first DATA statement in the program.
To read the same data repeatedly, place this statement before the READ statement.
If you supply "line number," the DATA statement specified by the number will be read independent of its location in the program.

Example:

```
LIST

10 READ A,B,C,D
20 DATA 1,2,3,4
30 RESTORE
40 READ E
50 PRINT A+B+C+D+E

RUN
 11
Ready
```

Statement SCREEN

Function: Controls the active and the visual windows

Format: SCREEN active window, visual window

Description: SC-3000 has two independent windows.

1: Text window for program input
2. Graphics window for graphics display

The BASIC interpreter initializes both of the windows to 1:

SCREEN 1, 1

You must execute, prior to any graphics commands

SCREEN 2, 2

The active window is utilized by the PRINT statement and so on, while the visual window is for graphics output.
The CLS statement erases the active window implied by the SCREEN statement.

Example:

```
SCREEN 2,2:CLS
Ready
```

Statement SOUND

Function: Generates sounds having given frequencies

Format: SOUND channel, frequency, volume

Usage SOUND 1, 1000, 15 [CR]
Ready

Description: (Channel)
Each channel corresponds to a certain fixed tune.
By mixing the first three channels, you can play a trio.

Channel	Function
0	Turn off the sound
1	Generate ordinary notes
2	Generate ordinary notes
3	Generate ordinary notes control frequency when the channel specified is 4 or 5
4	Generate white noises
5	Generate synchronized noises

(Frequency)
Specify desired frequency if the channel selected is 1, 2 or 3.
If the selected channel is 4 or 5, specify one of the integer among 0 thru 3 according to the following description:

0 thru 2: Each corresponds to a predetermined frequency
3: Frequency is controlled by the channel 3

(Volume)

0: Switch off the sound
1: Minimum volume
≀
15. Maximum volume

With this statement you can produce amusing sound effects to your games or compose and produce melodies. See the following table.

Example:

```
LIST

10 RESTORE 80
20 READ D
30 IF D=0 THEN SOUND0:END
40 SOUND 1,D,15
50 SOUND 2,D*2,11
60 SOUND 3,D*3,9
70 GOTO 20
80 DATA 370,370,392
81 DATA 440,440,392
82 DATA 370,330,294
83 DATA 294,330,374
84 DATA 370,330,330
85 DATA 370,370,392
86 DATA 440,440,392
87 DATA 370,330,294
88 DATA 294,330,370
89 DATA 330,294,294,0
```

This program makes use of synchronized noises.
The channel 3 controls frequency while the channel 5 controls volume.

```
LIST

10 FOR I=1500 TO 3000 STEP 10
20 SOUND 3,1,0
30 SOUND 5,3,15-ABS(1/100-20)
40 NEXT I
50 SOUND0
```

ORGAN

```
LIST

10 REM ----- doremi ---
20 CLS
30 PRINT" #la   #do re   #fa so la   #do re
"
40 PRINT"  W     R  T     U  I  O     @  [ "
50 PRINT
60 PRINT" A  S  D  F  G  H  J  K  L  ;  :  ]"
70 PRINT" la ti do re mi fa so la ti do re mi
"
80 Z$=INKEY$
90 IF Z$="A" THEN SOUND1,220,15
100 IF Z$="W" THEN SOUND1,233,15
110 IF Z$="S" THEN SOUND1,247,15
120 IF Z$="D" THEN SOUND1,262,15
130 IF Z$="R" THEN SOUND1,277,15
140 IF Z$="F" THEN SOUND1,294,15
150 IF Z$="T" THEN SOUND1,311,15
160 IF Z$="G" THEN SOUND1,330,15
170 IF Z$="H" THEN SOUND1,349,15
180 IF Z$="U" THEN SOUND1,370,15
190 IF Z$="J" THEN SOUND1,392,15
200 IF Z$="I" THEN SOUND1,415,15
210 IF Z$="K" THEN SOUND1,440,15
220 IF Z$="O" THEN SOUND1,466,15
230 IF Z$="L" THEN SOUND1,494,15
240 IF Z$=";" THEN SOUND1,523,15
250 IF Z$="@" THEN SOUND1,554,15
260 IF Z$=":" THEN SOUND1,587,15
270 IF Z$="[" THEN SOUND1,622,15
280 IF Z$="]" THEN SOUND1,659,15
290 IF Z$="" THEN SOUND0
300 GOTO 80
```

Frequency Table

Notes			f1	f2	f3	f4	f5
C	do	f		131	262	523	1047
C#, D♭				139	277	554	1109
D	re	g		147	294	587	1175
D#, E♭				156	311	622	1245
E	mi	a		165	330	659	1319
F	fa	b		175	349	698	1397
F#, G♭				185	370	740	1480
G	so	c		196	392	784	1568
G#, A♭				208	415	831	1661
A	la	d	110	220	440	880	1760
A#, B♭			117	233	466	932	
B	si	e	123	247	494	988	

Unit: Hz

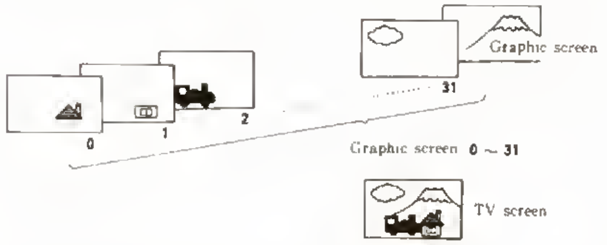
Statement SPRITE

Function: Moves sprite patterns on the screen.

Format: SPRITE sprite window, (X, Y), sprite name, color code.

Description: This statement is used to move figures constructed by the PATTERN statement to the specified coordinate on the screen to construct a sprite pattern.
No re-definition of patterns is needed.
The arguments to the statement are as follows:

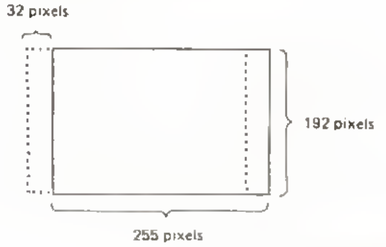
Sprite window
An integer in the range 0 thru 32 each corresponding to one sprite window.
A window with a lower id number is placed in front of a window with a higher id number.



Sprite name:
This is the number you supplied to the PATTERN statement.
A sprite name can be used on more than one sprite windows
The number of sprite patterns can be up to 56 and 32 of them can be simultaneously displayed onto the screen.

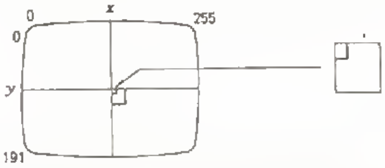
Color code:
One sprite has one color.

Coordinate.
If you move sprites rightward with the horizontal range exceeding 255, they will reappear from the left border of the screen, and this is the intended result.
Remind this fact when moving figures horizontally
In case a sprite moves leftward beyond the left margin of the screen, the sprite window is shifted to left by the amount of 32 pixels and the sprite is entirely erased from the screen.
In this case you are recommended to set a left margin in your coordinate since continuing the move will cause the "Parameter error".



Note: The dotted line indicates the sprite window shifted by the EC (Early Clock) bit

The origin of the 8-by-8 frame regarded as a coordinate for character patterns is on the uppermost, extreme left, which is also the case for sprite windows



The SPRITE statement has the nice characteristic of being able to put figures in front of or on the back of others by drawing them on different sprite windows.
A sense of perspective can easily be introduced into your graphics pictures by utilizing this characteristics intelligently.

Note 1: Although you can display up to 32 sprite windows simultaneously, only up to 4 windows can be placed on the same horizontal line (rastor) and the 5th window, if supplied, becomes invisible blocked by the former windows. But since windows are blocked not by sprites, but by dots, if you move the 5th window vertically, the window will begin to be blocked and reappear dot-wise on the screen.

Note 2: Although each sprite window can have one color, you can combine from 2 to 4 windows to create a, say, 4-colored character.
In case you must put multiple number of sprite windows on a same horizontal line, plan them carefully reminding the note 1 above.

Example:

```
10 M=1
20 SCREEN 2,2:CLS
30 MAG M:C=RND(1)*13+1
40 CURSOR 10,10:PRINT CHR$(17);"MAG";M
50 FOR Y=0 TO 191 STEP 4
60 PATTERN S#0,"00193F3C1C0D0F7B"
70 PATTERN S#I,"0C0F0F0F07031B07"
80 PATTERN S#2,"00CCFE9E9CD878EC"
90 PATTERN S#3,"1AFAF8F0EC7C3800"
100 Y1=Y:GOSUB 190
I10 PATTERN S#0,"00193F3CIC0D0F1B"
I20 PATTERN S#1,"2C2F0F07IB1F0E00"
130 PATTERN S#2,"00CCFE9E9CD878EF"
140 PATTERN S#3,"18F8F8F8F0606C70"
I50 YI=Y+2:GOSUB 190
160 NEXT Y
170 M=M+2:IF M 3 THEN M=1
I80 GOTO 20
190 SPRITE 0,(120,YI),0,C
200 SPRITE 0,(120,Y1+1),0,C
210 RETURN
```

See Also: PATTERN, MAG

Statement STOP

Function: Interrupt execution of a program for a while

Format: STOP

Description: Insert this statement into a program that behaves other than you expect. It will temporarily suspend execution of the program right where it was inserted.
By inserting this statement to various part of the program, you can keep an eye on the intermediate results step by step, and thus can find out what's wrong with the program.
For example, typing

PRINT "variable name" CR

will show you the intermediate value of the variable indicated by "variable name."
If you interrupt a program with this statement, the message

Break in "line number"

will appear on the screen
The CONT statement will resume execution of the program right from immediately after the STOP statement if you didn't modify it.

Example:

```
LIST

10 FOR 1=1 TO 9
20 FOR J=1 TO 9
30 PRINT I*J;
40 NEXT J:PRINT
50 STOP
60 NEXT I

RUN
 1 2 3 4 5 6 7 8 9
Break in 50
PRINT 1,J
 1                     10
Ready
CONT
 2 4 6 8 10 12 14 16 18
Break in 50
```

Statement VERIFYM

Function: Compares machine language programs saved on cassette with the program in memory

Format: VERIFYM "filename", verify start address

Description: This statement compares the machine language program on cassette indicated by "filename" and the program in memory The message "Verify end" will be displayed if no difference has been detected between the two programs. If you specify the verify start address, the comparison will start from that address. If not, it will start from the address as previously indicated by CSAVEM If "filename" is omitted, comparison will be done between the program in memory and the first machine language program found on the cassette

Note: Filename must be the one you christened on save.

Example:

```
VERIFYM
* Verifying start
 Found OBJ: HEX DATA
* Verifying end
Ready
```

Statement VPOKE

Function: Writes data to the VRAM (Video RAM)

Format: VPOKE address, data

Description: By writing data into the VRAM, you can draw characters or figures on the screen. The same with the text window

Example:

```
10 FOR A=&H3C00 TO &H3FBF
20 VPOKE A,65
30 NEXT A
```

```
10 B=&H1800+32*8
20 FOR A=B TO B+7
30 VPOKE A,255
40 NEXT A
50 FOR C=0 TO 100:NEXT
60 FOR A=B TO B+7
70 VPOKE A,0
80 NEXT A
90 FOR C=0 TO 100:NEXT
100 GOTO 20
```

```
10 FOR V=15360 TO 15700
20 CURSOR 18,0:PRINT"VRAM ADDRESS"V
30 X=0:Y=10
40 VP=VPEEK(V)
50 VPOKE V+X+Y*40,VP
60 X=X+1:IF X=38 THEN X=0:Y=Y+1
70 NEXT V
```

Arithmatic Function ABS (absolute)

Function: Gives the absolute value for the arithmetic expression X

Format: ABS (X)

Description: The absolute value of a value is the same with the value if the value is positive, and is equal to the negative of the value if the value is negative

Example

```
PRINT ABS(-5)
 5
Ready
```

```
PRINT ABS(3*(-6))
 18
Ready
```

Arithmatic Function ACS (arc-cosine)

Function: Gives θ in COS (θ) inverse cosine function

Format: ACS (X) X must be in the range –1 thru 1

Example:

```
10 FOR S=-1 TO 1 STEP .5
20 X=ACS(S)
30 Y=DEG(X)
40 PRINT X,Y
50 NEXT S
RUN
 3.1415926536        180
 2.0943951024        120
 1.5707963268        90
 1.0471975512        60
 0                   0
```

Character String Function: ASC (ascii)

Function: Converts characters into corresponding numbers (ascii codes)

Format: ASC (character constant or character variable)
Only the first character of any string constant more than one character long will be converted

Description: Computers don't understand characters and symbols as the way human beings do. They only understand numbers.
The way they understand characters and symbols, they have a set of numbers ranging from 32 thru 255 within them, each number corresponding to each character and symbol on your keyboard
In this way they can distinguish the character A (which is 65 from the computer's point of view) from the character B (66)
Even though you supply a character string more than one character long, as in

? ASC ("BA")

this function only outputs the number corresponding to the first character of the string and ignores the rest.

Example Sort names in alphabetical order according to the first character of the names
On RUN, displays corresponding ascii codes to the input characters and symbols

PRINT ASC ("A") CR
 65 ← the ascii code for 'A' is 65
PRINT ASC ("!") CR
 33 ← the ascii code for '!' is 33

```
10 INPUT A$
20 O=ASC(A$)
30 PRINT O
40 GOTO 10
```

See also: (CHR$)

SORT

```
10 INPUT "number of DATA":N
20 DIM A$(N)
30 FOR I=1 TO N:READ A$(I):NEXT I
40 D=N
50 D=INT(D/2)
60 IF D<1 THEN 180
70 DD=N-D
80 FOR K=1 TO DD
90 J=K
100 IF ASC(A$(J)) <=ASC(A$(J+D)) THEN 160
110 N$=A$(J)
120 A$(J)=A$(J+D)
130 A$(J+D)=N$
140 J=J-D
150 IF J>=1 THEN 100
160 NEXT K
170 GOTO 50
180 FOR I=1 TO N:PRINT I,A$(I):NEXT I
200 DATA SUN,MERCURY,VENUS,EARS
210 DATA MOON,MARS,JUPITER,SATURN
220 DATA URANUS,NEPTUNE,PLUTO
230 DATA ASTEROID,MILKY WAY,GALAXY
RUN
number of DATA14
 1                   ASTEROID
 2                   EARS
 3                   GALAXY
 4                   JUPITER
 5                   MARS
 6                   MOON
 7                   MILKY WAY
 8                   MERCURY
 9                   NEPTUNE
 10                  PLUTO
 11                  SUN
 12                  SATURN
 13                  URANUS
 14                  VENUS
Ready
```

Arithmatic Function ASN (arc-sine)

Function: Gives θ in SIN (θ) inverse sine function

Format: ASN (X) X must be in the range –1 thru 1
The value of ASN (X) is in radian

Example :

```
10 FOR S=-1 TO 1 STEP .5
20 X=ASN(S)
30 Y=DEG(X)
40 PRINT X,Y
50 NEXT S
RUN
-1.5707963268        -90
-.5235987756         -30
 0                    0
 .5235987756          30
 1.5707963268         90
Ready
```

Arithmetic Function	ATN	(arc tangent)

Function: Gives the inverse tangent   inverse tangent function

Format: ATN (X)

This function returns values within the range $-\frac{\pi}{2}$ thru $\frac{\pi}{2}$

Example :

```
10 X=ATN(1)
20 Y=DEG(X)
30 PRINT X,Y
RUN
 .7853981634          45
```

Character String Function	CHR$	(character $)

Function. Converts ascii codes into corresponding characters or control codes

Format. CHR$ (arithmetic expression to be converted)

Argument must be an integer in the range 32 thru 255

Floating point numbers are trancated to integers

Description: Each character and symbol in your computer has associated with it a code number. And this function is used to convert a code number to its corresponding character or symbol.

Computers can perform numeric comparison, or can sort strings in alphabetical order because they have a fixed set of code numbers.

See the Character Code Table in the appendix. Characters and symbols are assigned code numbers greater than or equal to 32

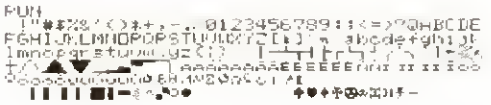
CHR$ function can be used for getting control codes too.

Usage. PRINT CHR$ (65)

A ← the ascii code 65 corresponds to the character 'A'

Example: Let's peak in the characters stored in your computer

```
10 FOR M=32 TO 255
20 PRINT CHR$(M);
30 NEXT M
```



```
RUN
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDE
FGHIJKLMNOPQRSTUVWXYZ[\]^ ~ abcdefghijk
lmnopqrstuvwxyz{|}
+/\▲▼
ÖÖÖ
 ▌▌▌▌█▌━                     ♠♥♦♣
```

You can see the characters and symbols on your keyboard displayed on the screen Those are the characters and symbols stored in your computer.

See the Character Code Table and verify that the code numbers on your screen and on the table are the same.

Arithmetic Function	COS	(cosine)

Function A trigonometric function: gives the cosine of the arithmetic expression X

Format. COS (X)   the argument X must be in radian

Usage: Let's find out the cosines of 0°, 30°, 60°, 90°:

Example :

```
10 FOR X=0 TO 90 STEP 30
20 A=COS(RAD(X))
30 PRINT X;TAB(10);A
40 NEXT X
RUN
 0          1
 30         .86602540379
 60         .50000000001
 90         0
Ready
```

Arithmetic Function	DEG	(degree)

Function: Gives the equivalent angle in degree of the arithmetic expression X in radian

Format: DEG (X)

Description This function is the inverse of rbe RAD function and returns the equivalent angle in degree of the expression X (in radian) by multiplying it by 180/pr.

Example :

```
PRINT DEG(0.26)
 14.896902673
Ready
```

Arithmetic Function	EXP	(exponent)

Function. Gives powers of e (the base for the natural logarithm)

Format: EXP (X)

Usage: Let's calculate e^1, e^2, and e^3 respectively.

Example :

```
10 FOR I=1 TO 3
20 X=EXP(I)
30 PRINT"EXP(";I;")=";X
40 NEXT I

RUN
EXP( 1)= 2.7182818284
EXP( 2)= 7.3890560987
EXP( 3)= 20.085536923
Ready
```

General Function	FRE	(free)

Function: Gives the amount of free memory.

Format. FRE

Description. This function gives the amount of free area among the area available to the BASIC

Example :

```
PRINT FRE
 15000
Ready
```

Character String Function	HEX$	(hexa $)

Function: Converts values of numeric expressions into equivalent hexadecimal numeric character strings

Format. HEX$ (numeric variable or expression)

Description The range of the argument to this function is from -32768 thru 32767, and the decimal fraction, if any, is truncated

Computers handle numbers in hexadecimal as well as in decimal into the equivalent number in hexadecimal.

To convert a hexadecimal number into equivalent decimal number, type

PRINT &H "hexadecimal number"

To distinguish between decimal and hexadecimal numbers, add "/H' at the head of any hexadecimal numbers.

The hexadecimal number &H10 is equivalent to the decimal number 16.

Usage. Let's convert -10, -5, 0, 5, 10, 15 into equivalent hexadecimal numeric character strings:

Example:

```
10 FOR S=-10 TO 15 STEP 5
20 X$=HEX$(S)
30 PRINT S;"=";X$
40 NEXT S
RUN
-10=FFF6
-5=FFFB
 0=0
 5=5
 10=A
 15=F
Ready
```

See also ASC

Character String Function INKEY$ (in key $)

Function: Gives a character entered from the keyboard

Format: INKEY$

Description: Gives the character corresponding to the key being pushed at the time of execution of this function. If no key is being pushed, it gives the null string.
This funciton cannot detect the RESET, BREAK and the FUNC keys.
* Null string is the character string zero-character long, and is represented as two consecutive double quotes ("_").

Usage:

```
10 X$=INKEY$
20 IF X$="" THEN 10
30 PRINT X$;
40 GOTO 10
```

The line 20 keeps watching whether a key is being pushed. If no key is being pushed, X$ is assigned the null string (the character string with nothing in it) and nothing is displayed on the screen.
The program thus keeps looping between the lines 10 and 20 (an infinite loop). If a key is pushed at this time, X$ is assigned the character corresponding to the key and the line 30 displays it. To get out of this infinite loop, type

```
25 IF X$="Z" THEN 100
100 PRINT "END":END
```

Hit the Z key to end this program.

Example:

Operation can be started by ⊡ ⊡

```
LIST

10 X=18:Y=11:CLS
20 CURSOR X,Y:PRINT" "A" "
30 A$=INKEY$
40 IF A$="" THEN 30
50 IF A$=CHR$(28) THEN X=X+1
60 IF A$=CHR$(29) THEN X=X-1
70 IF X<0 THEN X=0
80 IF X>33 THEN X=33
90 GOTO 30
```

General Function INP (inport)

Function: Gives contents of the I/O area

Format: INP (address)

Description: This function gives data in the specified I/O port. In the following example, the data in the I/O port BE (hexa) is read into the variable A at line 10. The result may vary according to the current state of the computer.
I/O port numbers are predetermined by the system and must be in the range (&H00 thru &HFF).
This function is useful to detect, for example, the current state of the joystick which is an external I/O device.

Example:

```
10 FOR A=&HE0 TO &HE7
20 PRINT A;INP(A)
30 NEXT A

RUN
 224 128
 225 128
 226 208
 227 128
 228 0
 229 19
 230 195
 231 2
Ready
```

General Function INPUT$ (input dollar)

Function: Reads a character string having the specified lengsh from a sequential file indicat- by the file descriptor.

Format: INPUT$

Example:

```
10 OPEN"TEL NOTE" FOR INPUT AS #1
20 A$=INPUT$(20,#1)
30 PRINT A$
40 CLOSE #1
```

Arithmetic Function INT (integer)

Function: Trancates the decimal fraction of given arguments to return the resulting integer.

Format: INT (x)

Description: Use this function to set the greatest integer not greater than a given value or the value of an arithmetic expression.
This function is useful for numeric truncation or rounding up.

Example:

```
10 FOR N=1 TO 2 STEP 0.1
20 L=INT(N+0.5)
30 PRINT N;"=";L
40 NEXT N

RUN
 1= 1
 1.1= 1
 1.2= 1
 1.3= 1
 1.4= 1
 1.5= 2
 1.6= 2
 1.7= 2
 1.8= 2
 1.9= 2
 2= 2
Ready
```

Character String Function LEFT$ (left $)

Function: Gives a substring of the given character string having the given number of length counting from the left

Format: LEFT$ (character string, length)

Description: Gives a substring of the given character string having the specified number of length counting from the left. A space counts as one character.

Usage: Let's store into M$ the substring consisting of the first 6th characters of the character string stored in A$, and display it onto the screen.

Example:

```
10 A$="coffee cocoa mill"
20 M$=LEFT$(A$,6)
30 PRINT M$

RUN
coffee
Ready
```

Up to 6th characters counting from the left

See also RIGHT$

Character String Function LEN (length)

Function: Gives the number of characters in the given character string

Format: LEN (character string)

Description: This function gives the number of characters in the given character string. Special symbols and spaces count as one within a string enclosed by " ".

Usage: Let's count the number of characters in the character string contained in A$

```
10 A$="SEGA PERSONAL COMPUTER"
20 PRINT LEN(A$)
RUN
 22
Ready
```

Note that a space counted as one

Example:

```
LIST

10 A$="**********"
20 FOR I=1 TO LEN(A$)
30 PRINT LEFT$(A$,I)
40 NEXT I
RUN
*
**
***
****
*****
******
*******
********
*********
**********
Ready
```

Arithmetic Function LGT (log ten)

Function: Gives logarithm to the base 10 (common logarithm)

Format: LGT (X)

Usage: Let's find out the common logarithms of 10, 100 and 1000.

Example:

```
10 N=1
20 N=N*10
30 X=LGT(N)
40 PRINT"LGT(";N;")=":X
50 IF N 1000 THEN 20
RUN
LGT( 10)= 1
LGT( 100)= 2
LGT( 1000)= 3
Ready
```

Arithmetic Function LOG (log)

Function: Gives logarithm to the base e (natural logarithm)

Format: LOG (X)

Example:

```
10 FOR J=1 TO 3
20 X=LOG(J)
30 PRINT "LOG(":J:")=";X
40 NEXT J
RUN
LOG( 1)= 2.67468532E-11
LOG( 2)= .69314718057
LOG( 3)= 1.0986122886
Ready
```

Arithmetic Function LTW (log two)

Function: Gives logarithm to the base 2

Format: LTW (X)

Usage: See LOG.

Example:

```
10 FOR J=2 TO 10 STEP 2
20 X=LTW(J)
30 PRINT"LTW(":J:")=";X
40 NEXT J
RUN
LTW( 2)= 1
LTW( 4)= 2
LTW( 6)= 2.5849625007
LTW( 8)= 3
LTW( 10)= 3.3219280949
Ready
```

Character String Function MIDS (mid $)

Function: Gives a substring of the given character string

Format: MIDS (character string, *m*, *n*)
This gives the substring of the character string starting from the *m*-th character and *n*-characters long

Description: This function gives a substring of given number of length taking from the given character string. A space counts as one character.

Usage: Let's get the substring 5th-characters long starting from the 8th character of the character string.

```
10 A$="coffee cocoa milk"
20 M$=MID$(A$,8,5)
30 PRINT M$       ↑  └──────── length
RUN               │
cocoa        Starting character
Ready
```

Length can be omitted as in MIDS (character string, *m*), in which case the result is the substring starting from the *m*-th character up to the end of the character string

Example:

```
PRINT MID$("ABCDEFGHI",5)
EFGHI
Ready
```

See also LEFTS, RIGHTS

General Function PEEK (peek)

Function: Gives content of memory at specified address.

Format: PEEK (address)

Description: Use this function to peek at the content of memory at the desired address. The address must be in the range 0 thru 65535 (&H0 thru &HFFFF). The value is returned as one byte integer

Example:

```
10 FOR A=&H8000 TO %H8FFF STEP 8
20 PRINT RIGHT$("000"+HEX$(A),4);
30 FOR B=A TO A+7
40 C=PEEK(B)
50 PRINT " ";RIGHT$("0"+HEX$(C),2);
60 NEXT B:PRINT
70 NEXT A
```

Arithmetic Function PI (pi)

Function: Gives the ratio of circumference of circle to diameter

Format: PI

Description: 3.1415926536 is assigned to PI in your computer.

Example:

```
PRINT PI
 3.1415926536
Ready


LIST

10 INPUT"radius ";A
20 S=A*A*PI
30 PRINT "area of circle ";S

RUN
radius 5
area of circle 78.53981634
Ready
```

Arithmetic Function RAD (radian)

Function. Gives the equivalent angle in radian of the arithmetic expression X in degree

Format RAD (X)

Description. This function gives the equivalent angle in radian of the expression X (in degree) by multiplying it by pi/180

Usage. Let's convert 0°, 15°, 30°, 45°, 60° into equivalent angles in radian:

Example:

```
LIST

10 FOR I=0 TO 60 STEP 15
20 X=RAD(I)
30 PRINT"RAD(";I;")=";X
40 NEXT I

RUN
RAD( 0)= 0
RAD( 15)= .2617993878
RAD( 30)= .5235987756
RAD( 45)= .7853981634
RAD( 60)= 1.0471975512
Ready
```

Character String Function RIGHT$ (right $)

Function: Gives a substring of the given character string having the given number of length counting from the right

Format. RIGHT$ (character string, length)

Description: This function gives a substring of the given character string having the specified number of length counting from the right of the given string. A space counts as one character

Usage: Let's store into M$ the substring four-characters long (counting spaces) counting from the right of the character string stored in A$, and display it onto the screen:

Example:

```
10 A$="coffee cocoa milk"
20 M$=RIGHT$(A$,4)
30 PRINT M$
RUN
milk
Ready
```

Up to 4th characters counting from the right

See also. LEFT$, MID$

Arithmetic Function RND (random)

Function Generates random numbers

Format: RND (x)
where
- x > 0 : Generate random numbers successively
- x = 0: Initialize the random number series
- x < 0 : Permute the random number series

Description: The random numbers generated with this function are fractions greater than or equal to 0 and less than 1. Multiply them by appropriate factor to set random numbers having desired decimal places.

Example 1. Let's generate random numbers successively

The random numbers have 11 decimal places.

To set a sequence of random numbers in the range 1 thru 500, multiply the returned values by 500 and add 1 to them.

If you need random numbers in whole number, use the INT statement to discard the decimal fraction.

```
10 FOR N=1 TO 5
20 R=RND(1)
30 PRINT R
40 NEXT N

RUN
 .74622984084
 .01811711048?
 .51872376941
 .86514622469
 .35638778417
Ready
```

R = INT (RND (1) * 500 + 1)

```
10 FOR N=0 TO 5
20 R=INT(RND(1)*500+1)
30 PRINT R;:NEXT N

RUN
 173 450 334 145 6 227

Ready
```

Example 2. Let's generate random numbers in the range 0 thru 5 and add 1 to them.

```
10 FOR N=1 TO 6
20 R=INT(RND(-1)*6+1)
30 PRINT R;
40 NEXT N

RUN
 1 6 4 3 2 4

Ready
```

You see we have generated random numbers in the range 1 thru 6. You can us this as a fake dice

See Also: INT

Arithmatic Function SGN (sign)

Function: Used to get the sign of numeric expressions/values

Format: SGN (X) returns –1 if X is negative
0 if X is 0
1 if X is positive

Example

```
LIST

10 FOR I=-2 TO 2
20 N=SGN(I)
30 PRINT"SGN(";I;")=";N
40 NEXT I

RUN
SGN(-2)=-1
SGN(-1)=-1
SGN( 0)= 0
SGN( 1)= 1
SGN( 2)= 1
Ready
```

Arithmetic Function SIN (sine)

Function. A trigonometric function. gives the sine of the arithmetic expression X

Format: SIN (X) the argument X must be in radian

Usage Let's find out the sines of 0°, 30°, 60°, 90°.

Example:

```
10 FOR TH=0 TO 90 STEP 30
20 S=SIN(RAD(TH))
30 PRINT TH;TAB(10);S
40 NEXT TH

RUN
 0         0
 30        .5
 60        .86602540379
 90        1
```

Character String Function SPC (space)

Function: Gives specified number of consecutive spaces

Format: SPC (length)

Description This function must be used in one of the PRINT, LPRINT, or PRINT# statements

The length must be an integer in the range 0 thru 55. In case length as a value has a decimal fraction, the fraction will be truncated entirely

Usage:

```
10 PRINT"ABC";SPC(10);"XYZ"
RUN
ABC ←——————→ XYZ
Ready
          10 spaces
```

If some characters happen to be in the range specified to SPC, these characters will be erased from the screen.

See also TAB

Arithmetic Function SQR (square root)

Function Gives square roots of given valves

Format SQR (X)

Usage Let's calculate $\sqrt{2}$ and $\sqrt{3}$

Example :

```
10 INPUT "FIGURE ";A
20 X=SQR(A)
30 PRINT"SQR.ROOT";A;"=";X
40 GOTO 10
RUN
figure 3
sqr.root 3 = 1.7320508075
figure 5
sqr.root 5 = 2.2360679775
figure
Break in 10
```

General Function STICK

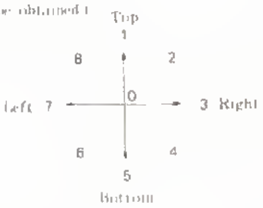
Function Gives directions of joystick

Format STICK (*f*)

Description

(Value to be obtained)

Parameter (*f*) 1 = Joystick 1
2 Joystick 2



Example :

```
10 REM JOY STICK TEST
20 B$="SHOOT":CLS
30 P1=STICK(1):P2=STICK(2)
40 S1=STRIG(1):S2=STRIG(2)
50 FI$="":F2$=""
60 IF P1=1 THEN F1$="UP     "
70 IF PI=3 THEN F1$="RIGHT  "
80 IF P1=5 THEN F1$="DOWN   "
90 IF P1=7 THEN F1$="LEFT   "
100 IF P2=1 THEN F2$="UP     "
110 IF P2=3 THEN F2$="RIGHT  "
120 IF F2=5 THEN F2$="DOWN   "
130 IF P2=7 THEN F2$="LEFT   "
140 IF S1 0 THEN F1$=F1$+B$+STR$(SI)
150 IF S2 0 THEN F2$=F2$+B$+STR$(S2)
160 CURSOR 10,10:PRINT CHR$(5)
170 CURSOR 1,10:PRINT"PLAYER 1 ";F1$
180 CURSOR 10,15:PRINT CHR$(5)
190 CURSOR 1,15:PRINT"PLAYER 2 ";F2$
200 GOTO 30
```

Character String Function STR$

Function: Converts numeric values to equivalent numeric character strings

Format: STR$ (numeric expression)

Description: Use this function to convert numeric values to equivalent numeric character strings. Note that the result cannot be used in numeric calculations.

Usage: If the value is a positive number, a space is added at the head of the result:

```
LIST

10 A=1:B=3
20 D$=STR$(A)+STR$(B)
30 D=A+B
40 PRINT D$,D
```

```
RUN
 1 3                        4
```

Result of the line numbered 20 Result of the line numbered 30

Note STR$ (A) turned the given number into the equivalent numeric character string. Adding characters doesn't do any calculation but putting them together.

Example

```
LIST

10 A=1
20 A=A*10
30 IF A 1000 THEN END
40 D$=RIGHT$("   "+STR$(A),4)
50 PRINT D$
60 GOTO 20
RUN
  10
 100
1000
Ready
```

See also VAL

General Function STRIG (stick trigger)

Function: Gives states of the joystick trigger (the push button)

Format: STRIG (*f*)

Description:

(Value to be obtained)

Parameter (*f*) 1 Joystick 1
2 Joystick 2

0 off
1 Trigger (left) ON
2 Trigger (right) ON
3 Trigger (left, right) ON

Example :

```
10 REM JOY STICK TEST
20 B$="SHOOT":CLS
30 P1=STICK(1):P2=STICK(2)
40 S1=STRIG(1):S2=STRIG(2)
50 FI$="":F2$=""
60 IF PI=1 THEN F1$="UP     "
70 IF PI=3 THEN F1$="RIGHT  "
80 IF P1=5 THEN F1$="DOWN   "
90 IF P1=7 THEN F1$="LEFT   "
100 IF P2=1 THEN F2$="UP     "
110 IF P2=3 THEN F2$="RIGHT  "
120 IF P2=5 THEN F2$="DOWN   "
130 IF P2=7 THEN F2$="LEFT   "
140 IF S1 0 THEN FI$=FI$+B$+STR$(S1)
150 IF S2 0 THEN F2$=F2$+B$+STR$(S2)
160 CURSOR 10,10:PRINT CHR$(5)
170 CURSOR 1,10:PRINT"PLAYER 1 ";FI$
180 CURSOR 10,15:PRINT CHR$(5)
190 CURSOR 1,15:PRINT"PLAYER 2 ";F2$
200 GOTO 30
```

Character String Function TAB (tabulation)

Function Specifies a position from the left hand side of the screen

Format TAB (number)

Description This function must be used in one of the PRINT, LPRINT, or PRINT# statements

The number must be in the range 0 thru 37, and if it has a decimal fraction, the fraction will be truncated entirely.

TAB (0) is equivalent to the extreme left hand side of the screen

Example

```
10 PRINT TAB(5);"ABC"
RUN
     ABC
```

5 spaces

The TAB function will not erase characters which happen to be in the range specified to it.

Remember this function since it is quite often used in the PRINT statement

See also SPC

Arithmetic Function TAN (tangent)

Function. A trigonometric function: gives the tangent of the arithmetic expression X

Format: TAN (X) the argument X must be in radian

Usage Let's find out the tangents of values (in degree) input from the keyboard.

Example :

```
10 INPUT "angle ";30
20 X=tan(A)
30 PRINT"TAN(";a;" )=";x
RUN
angle 30
TAN( 30 )= .57735026919
```

Character String Function TIMES

Function: Sets and displays the inner clock

Format TIMES

Description Each computer has a clock in it.
And a very accurate, digital quartz clock is contained in your computer.
As soon as you switch on the power of your computer, the clock begins to tick with the interval of 1 second.
At the time of power-on, the clock is set to 00:00:00. Typing

PRINT TIMES [CR]

will display the time elapsed from the time of power-on

Example :

For example, to set the time to 8 15.00, you type

TIMES = "08 15·00" [CR]

the computer will answer

Ready

```
10 CLS
20 IF TIME$=T$ THEN 20
30 CURSOR 10,5:PRINT TIME$
40 T$=TIME$:BEEP
50 GOTO 20
```

Once you set the time, the clock keeps on going until you push the RESET button or turn off the power Thus you can use your computer as a digital clock

Character String Function VAL (value)

Function Converts numeric character strings into equivalent numeric values

Format VAL (numeric character string)

Description The first character of the argument to this function must be either numeric or one of '+', '-', '$' or a space. Otherwise the result will be 0.
If the argument string contains a non-numeric character, then the rest of characters starting from that character will be ignored.
This is the inverse function of STR$

Example :

```
10 A$="12345"
20 B$="11111"
30 C$=A$+B$                    ← Addition of numeric character strings
40 C=VAL(A$)+VAL(B$)  ← Addition of numeric values
50 PRINT C$
60 PRINT C
RUN
1234511111      ← Numeric character string
 23456          ← Numeric value
Ready
```

See also STR$

General Function VPEEK

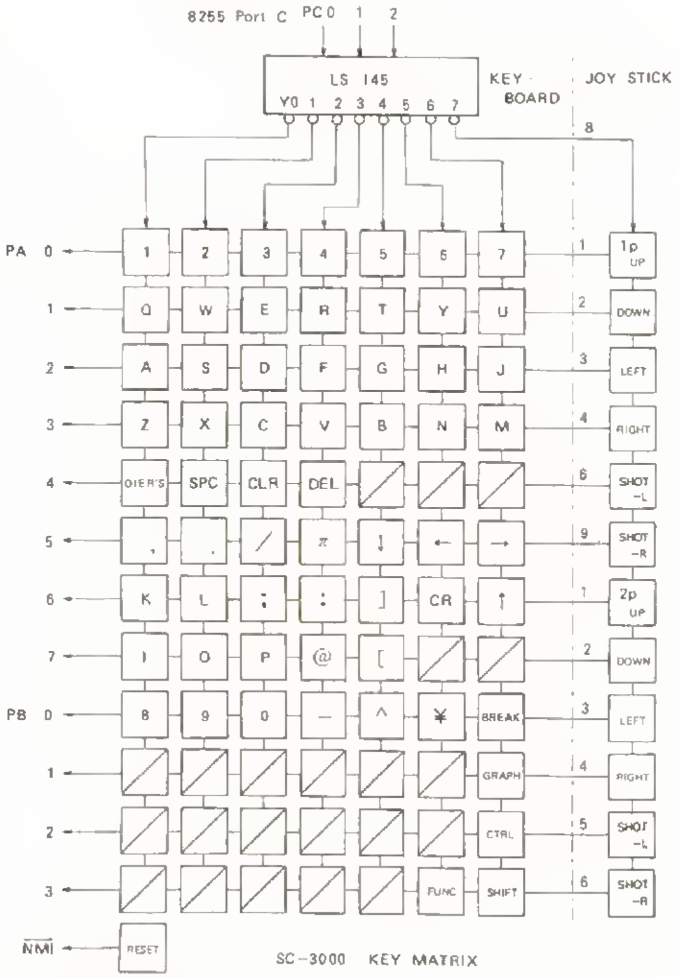
Function: Gives the contents of the Video RAM

Format: VPEEK (Video RAM address)

Description: This function reads from the VRAM data for the characters or figures currently isplayed on the screen Refer to the address map for the URAM.

Example :

```
10 FOR V=15360 TO 15780
20 X=0:Y=10
30 VP=VPEEK(V)
40 VPOKE V+X+Y*40,VP
50 X=X+1:IF X=38 THEN X=0:Y=Y+1
60 NEXT V
```



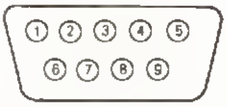
SC–3000 KEY MATRIX

1. The Joystick Terminal

Two connector sockets, JOY1 and JOY2, for joysticks are supplied at the rear end of the SF-3000, each connected to the P P.I inside the SF-3000.

Pin number	Function	
1	Lever	UP
2		DOWN
3		LEFT
4		RIGHT
5	———	
6	Trigger LEFT	
7	———	
8	Common	
9	Trigger RIGHT	

AMP 9-pin male type

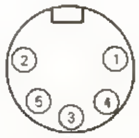


Outlook of the joystick connector socket and its pin numbers

2. Video/audio Signal Output Terminal

Pin number	Function
1	Audio signal
2	Video signal
3	GND
4	GND
5	GND

DIN 5-pin female type

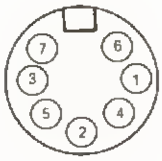


Outlook of the VIDEO connector socket and its pin numbers

3 Serial Printer Connector Tip

Pin number	Function
1	$\overline{\text{FAULT}}$
2	BUSY
3	DATA
4	$\overline{\text{RESET}}$
5	$\overline{\text{FEED}}$
6	GND
7	NC

DIN 7-pin female type



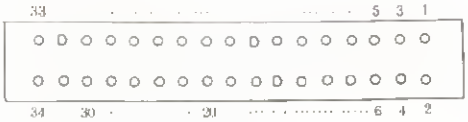
Outlook of the serial printer connector socket and its pin numbers

SC-3000 Extention Slot Connector Terminal

Side A (lower)		Side B (upper)	
Pin number	Signal name	Pin number	Signal name
1	A_0	1	+5V
2	A_1	2	+5V
3	A_2	3	$\overline{\text{DSRAM}}$
4	A_3	4	$\overline{\text{CEROM2}}$
5	A_4	5	$\overline{\text{MEMR}}$
6	A_5	6	$\overline{\text{MEMW}}$
7	A_6	7	$\overline{\text{I/OR}}$
8	A_7	8	$\overline{\text{I/OW}}$
9	A_8	9	N.C.
10	A_9	10	$\overline{\text{MREQ}}$
11	A_{10}	11	CON
12	A_{11}	12	$\overline{\text{RAS1}}$
13	A_{12}	13	$\overline{\text{CAS1}}$
14	A_{13}	14	RAM A7
15	D_0	15	$\overline{\text{RAS2}}$
16	D_1	16	$\overline{\text{CAS2}}$
17	D_2	17	MUX
18	D_3	18	A_{14}
19	D_4	19	A_{15}
20	D_5	20	N C
21	D_6	21	GND
22	D_7	22	GND

$A_0 - A_{15}$	CPU address bus
$D_0 - D_7$	CPU data bus
$\overline{\text{DSRAM}}$	OPEN if S-RAM for SC-3000 is used, +5V if not
$\overline{\text{CEROM2}}$	Select memory. 0 – 7FFF
$\overline{\text{MEMR}}$	CPU signal $\overline{\text{MREQ} \cdot \text{RD}}$
$\overline{\text{MEMW}}$	CPU signal $\overline{\text{MREQ} \cdot \text{WR}}$
$\overline{\text{1/OR}}$	CPU signal $\overline{\text{IORQ} \cdot \text{RD}}$
$\overline{\text{I/OW}}$ ······	CPU signal $\overline{\text{IORQ} \cdot \text{WR}}$
$\overline{\text{MREQ}}$	CPU signal
CON	······ P.P.I PB4
$\overline{\text{RAS 1}}$, $\overline{\text{CAS 1}}$ (8000 ~ BFFFH)	D–RAM Control signal
$\overline{\text{RAS 2}}$, $\overline{\text{CAS 1}}$ (C000 ~ FFFFH)	D–RAM Control signal
$\overline{\text{MUX}}$ ······	D–RAM Control signal

Connector socket to the computer

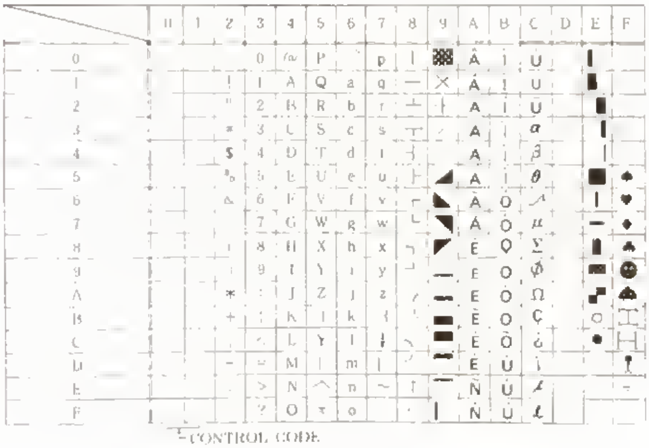


Pin number	Signal name	Pin number	Signal name
1	A_0	2	A_1
3	A_2	4	A_3
5	A_4	6	A_5
7	A_6	8	A_7
9	A_8	1 0	A_9
1 1	A_{10}	1 2	A_{11}
1 3	A_{12}	1 4	A_{13}
1 5	A_{14}	1 6	A_{15}
1 7	D_0	1 8	D_1
1 9	D_2	2 0	D_3
2 1	D_4	2 2	D_5
2 3	D_6	2 4	D_7
2 5	GND	2 6	$\overline{\text{I/OR}}$
2 7	GND	2 8	$\overline{\text{I/OW}}$
2 9	GND	3 0	$\overline{\text{MEMR}}$
3 1	GND	3 2	$\overline{\text{MEMW}}$
3 3	GND	3 4	$\overline{\text{MREQ}}$

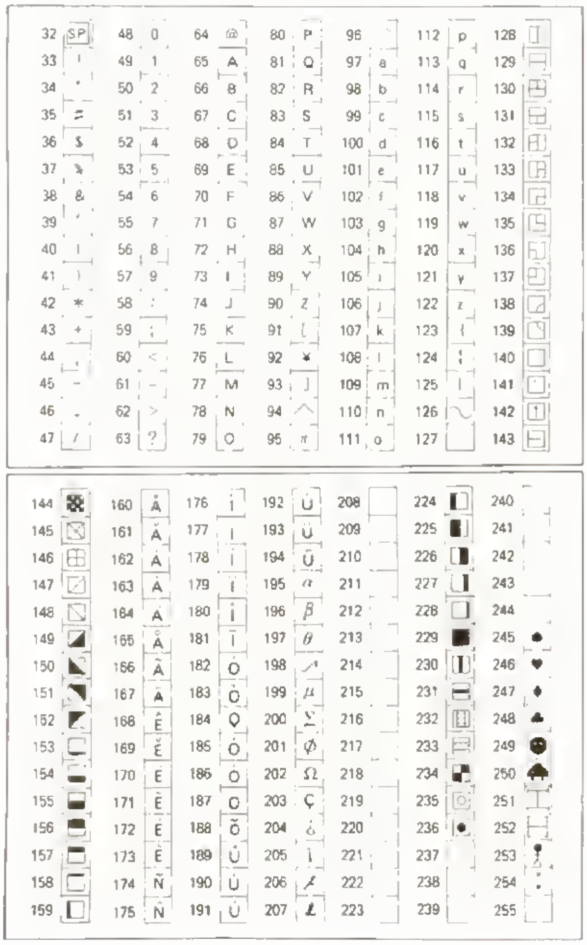
LIMITATIONS

Contents	Limitation
Characters taken into the inside from the screen	256 characters
Character numbers usable for actual text image by reserved words converted from line buffer	256 characters
Character numbers which can be handled as character string	255 characters
Level number such as operator priority etc	32 levels
Area for string operation	300 characters
FOR ~ NEXT nesting level number	16 levels
GOSUB, RETURN nesting level number	8 levels

CHARACTER SET


CONTROL CODE

CHARACTER CODE



Error Message

Message	Meaning
Array name	Argument to DIM statement not array
Bad file mode	Attempted to process file against its mode
Bad file number	Wrong file descriptor
Command parameter	Wrong argument to command
Can't continue	Can't continue with CONT statement
Device not ready	Printer not connected or out of order
Disk full	No free space in disk, can't SAVE
Disk I/O	Attempted to write on write-protected disk (protect hole open), invalid I/O
Disk offline	Attempted to access disk when offline (not inserted into drive)
Division by zero	Denominator was zero
Extra ignored	Wrong input data to INPUT statement, extra data ignored
File already exists	File already exists under the name supplied as argument to NAME (rename)
File already open	Attempted to open a file which is already open
File not found	There is no file as indicated
File not opened	Attempted to access file that is not open
File write protected	Write attempt to read-only file
FOR nesting	More than 8 levels of depth in FOR–NEXT
FOR variable name	Non numeric-variable to FOR (character type or array)
Function buffer full	Attempted to define more than 8 user functions
Function paremeter	Wrong argument to a function
GOSUB nesting	Nested subroutines with more than 16 levels
Illegal direct	Cannot execute direct statement
Illegal line number	Line number is improper
Input past end	Attempted to read sequential file past end of file
Line image too long	Too many characters in one line (RENUM, etc.)
Line number over	Line numbers exceeded 65535 in AUTO or RENUM
Number of subscripts	Number of subscripts is unusual

Message	Meaning
NEXT without FOR	No corresponding FOR to the NEXT statement
N-formula too complex	Too complex numeric expression
No program	Attempted to SAVE while no program in text buffer
Out of data	READ tried to read from a DATA statement having no data in it
Out of memory	
Overflow	Numeric overflow in arithmetic expression or in value
PRT. 2 not selected	Attempted to take hard copy (HCOPY) of graphics window without switching to appropriate printer (I/O unit)
Redim'd array	Attempted to re-declare an array
Redo from start	Wrong data to INPUT statement need re-input
RETURN without GOSUB	RETURN statement executed having no corresponding GOSUB
S-formula too complex	String expression too complex
Stack overflow	Too many parentheses, figure too complex for PAINT, or a user function is recursively defined
Statement parameter	Wrong argument to statement
String too long	Character string was more than 255 characters long
Syntax	Syntax error
System	The BASIC interpreter made an error (not possible)
Type mismatch	Type mismatch between data and variable (value, string)
Undef'd array	Attempted to ERASE undefined array
Undef'd function	Attempted to call undefined user function
Undef'd line number	No line under line number (RENUM, GOTO, GOSUB, IF-THEN, RESTORE, RUN)
Unprintable	Unknown error
Value of subscript	Subscript wrong or out of range
Verifying	Error during comparison between program in memory and program on cassette

SAMPLE PROGRAM

VRAM address of test

```
10 REM --    VRAM address of text  -
20 FOR V=15360 TO 15700
30 CURSOR30,0:PRINT V
40 X=0: Y=10
50 VP=VPEEK(V)
60 VPOKE V+X+Y*40,VP
70 X=X+1:IF X=38 THEN X=0:Y=Y+1
80 NEXT V
```

VRAM address, graphcs SCREEN

```
10 REM -VRAM address, graphcs SCREEN -
20 SCREEN 2,2:CLS
30 PRINT"H"
40 SCREEN 1,1:CLS
50 AD=&H0400+8
60 FOR A=AD TO AD+7
70 DA=VPEEK(A)
80 PRINT HEX$(DA)
90 NEXT A
```

SPRITE HIT

```
10 REM --- SPRITE HIT ----
20 SCREEN 2,2 :CLS
30 PATTERNS#0,"3B9CDAFFDE9C0830"
40 PATTERNS#4,"0202022262A22418"
50 X=80:Y=90
60 LINE(163,0)-(163,90),10
70 MAG2
80 SPRITE 0,(X,Y),0,5
90 SPRITE 2,(150,88),4,8
100 X=X+1
110 S=INP(&HBF) AND &H20
120 IF S<>0 THEN GOTO 140
130 GOTO 80
140 CLS
150 CIRCLE(120,90),50,6,.3
160 CURSOR110,85:PRIN"HIT !"
170 FORW=0 TO 100:NEXT W
180 CLS:GOTO 20
```

Bar graph

```
10 REM         Bar graph    ----
20 CLS
30 FOR I=0 TO 5
40 PRINT CHR$(65+I);"=";
50 INPUT "number of 0 to 20 ";A(I)
60 IF A(I)<0ORA(I)>20 THEN 40
70 NEXT I
80 CLS:N=20:FOR Y=0 TO 20
90 IF YMOD5=1 THEN CURSOR0,Y:PRINT N;:
N=N-5
100 CURSOR3,Y:PRINT CHR$(128);
110 NEXT Y
120 FOR X=3 TO 36
130 CURSORX,21:PRINT CHR$(158);
140 NEXT X
150 FOR X=6 TO 35 STEP 7
160 CURSORX,22:PRINTCHR$(65+(X-6)/7);
170 FOR Y=20 TO 21-A((X-6)/7) STEP -1
180 IF A((X-6)/7)=0 THEN 200
190 CURSORX,Y:PRINT"*";
200 NEXTY
210 NEXTX
220 GOTO 220
```

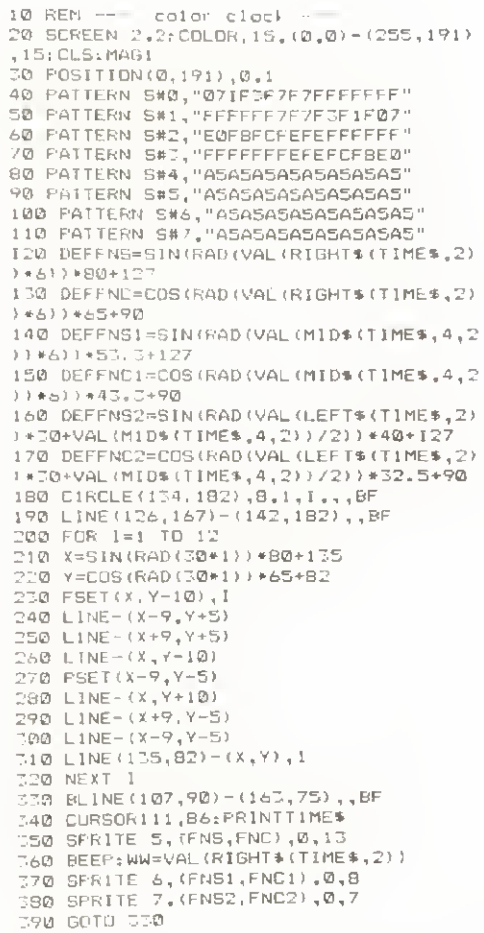
Line graph

```
10 REM ------ Line graph ------
20 CLS
30 FOR I=0 TO 7
40 PRINT CHR$(65+I);"=";
50 INPUT "number of 0 to 100 ";A(I)
60 IF A(I)<0ORA(I)>100 THEN 40
70 NEXT I
80 INPUT "Bar graph 1  or Line graph 2
 ";B
90 IF B<1ORB>2 THEN BEEP:GOTO 80
100 GOSUB 260
110 ON B GOTO 120,170
120 FOR X=41 TO 210 STEP 24
130 CURSORX,37:PRINTCHR$(65+(X-41)/24)
140 COLOR,4,(X-1,40)-(X+6,40+A((X-41)/
24))
150 NEXT X
160 GOTO 160
170 CURSOR41,37:PRINT "A"
180 CIRCLE(40,40+A(0)),2,4
190 PSET(40,40+A(0)),4
200 FOR X=64 TO 230 STEP 24
210 COLOR1:CURSORX,37:PRINTCHR$(65+(X-
64)/24+1)
220 CIRCLE(X,40+A((X-64)/24+1)),2,4
230 LINE-(X,40+A((X-64)/24+1))
240 NEXT X
250 GOTO 250
260 SCREEN 2,2:COLOR1,15:CLS
270 POSITION(0,191),0,1
280 LINE(31,170)-(31,40),1
290 LINE-(240,40)
300 CURSOR12,144:PRINT"100"
310 CURSOR18,94:PRINT"50"
320 CURSOR24,44:PRINT"0"
330 RETURN
```

COLOR AND SPRITE

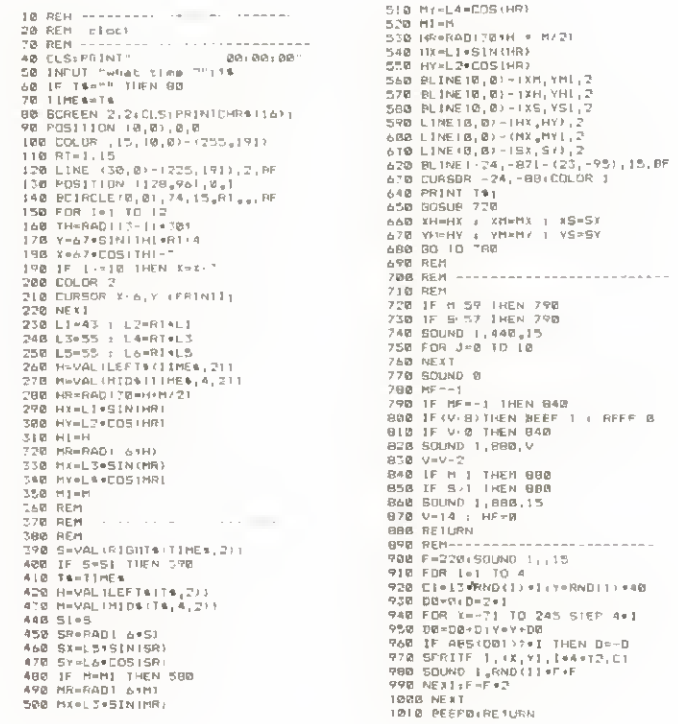
```
10 REM ---- COLOR  AND SPRITE -----
20 SCREEN 2,2:COLOR,1,(0,0)-(255,191),
1:CLS
30 PATTERN S#0,"FFFFFFFFFFFFFFFF"
40 PATTERN S#1,"FFFFFFFFFFFFFFFF"
50 PATTERN S#2,"FFFFFFFFFFFFFFFF"
60 PATTERN S#3,"FFFFFFFFFFFFFFFF"
70 PATTERN S#4,"A5A5A5A5A5A5A5A5"
80 PATTERN S#5,"A5A5A5A5A5A5A5A5"
90 PATTERN S#6,"A5A5A5A5A5A5A5A5"
100 PATTERN S#7,"A5A5A5A5A5A5A5A5"
110 FOR M=0 TO 1
120 RESTORE 190:MAG M:CLS
130 FOR I=0 TO 5.652 STEP .628
140 READ N,C
150 X=SIN(I)*80+127
160 Y=COS(I)*65+90
170 SPRITE N,(X,Y),0,C
180 NEXT I
190 DATA 0,2,31,0,1,4,30,13,2,10
200 DATA 29,3,7,7,28,6,4,5,27,11
210 FOR I=0 TO 6.28 STEP .1
220 X=SIN(I)*80+127
230 Y=COS(I)*65+90
240 SPRITE 9,(X,Y),4,15
250 NEXT I,M
260 GOTO 110
```

Color clock



```
10 REM --   color clock  --
20 SCREEN 2.2:COLOR,15,(0,0)-(255,191)
,15:CLS:MAG1
30 POSITION(0,191),0.1
40 PATTERN S#0,"071F3F7F7FFFFFFF"
50 PATTERN S#1,"FFFFFF7F7F3F1F07"
60 PATTERN S#2,"E0F8FCFEFEFFFFFF"
70 PATTERN S#3,"FFFFFFFEFEFCF8E0"
80 PATTERN S#4,"A5A5A5A5A5A5A5A5"
90 PATTERN S#5,"A5A5A5A5A5A5A5A5"
100 PATTERN S#6,"A5A5A5A5A5A5A5A5"
110 PATTERN S#7,"A5A5A5A5A5A5A5A5"
120 DEFFNS=SIN(RAD(VAL(RIGHT$(TIME$,2)
)*6))*80+127
130 DEFFNC=COS(RAD(VAL(RIGHT$(TIME$,2)
)*6))*65+90
140 DEFFNS1=SIN(RAD(VAL(MID$(TIME$,4,2
))*6))*53.3+127
150 DEFFNC1=COS(RAD(VAL(MID$(TIME$,4,2
))*6))*43.3+90
160 DEFFNS2=SIN(RAD(VAL(LEFT$(TIME$,2)
)*30+VAL(MID$(TIME$,4,2))/2))*40+127
170 DEFFNC2=COS(RAD(VAL(LEFT$(TIME$,2)
)*30+VAL(MID$(TIME$,4,2))/2))*32.5+90
180 CIRCLE(134,182),8,1,1,,,BF
190 LINE(126,167)-(142,182),,BF
200 FOR I=1 TO 12
210 X=SIN(RAD(30*I))*80+135
220 Y=COS(RAD(30*I))*65+82
230 PSET(X,Y-10),I
240 LINE-(X-9,Y+5)
250 LINE-(X+9,Y+5)
260 LINE-(X,Y-10)
270 PSET(X-9,Y-5)
280 LINE-(X,Y+10)
290 LINE-(X+9,Y-5)
300 LINE-(X-9,Y-5)
310 LINE(135,82)-(X,Y),1
320 NEXT I
330 BLINE(107,90)-(163,75),,BF
340 CURSOR111,86:PRINTTIME$
350 SPRITE 5,(FNS,FNC),0,13
360 BEEP:WW=VAL(RIGHT$(TIME$,2))
370 SPRITE 6,(FNS1,FNC1),0,8
380 SPRITE 7,(FNS2,FNC2),0,7
390 GOTO 330
```

clock



```
10 REM ---------   ---  ------
20 REM  clock
30 REM --------- -- ----------------
40 CLS:PRINT"              00:00:00"
50 INPUT "what time ?";T$
60 IF T$="" THEN 80
70 TIME$=T$
80 SCREEN 2,2:CLS:PRINTCHR$(16);
90 POSITION (0,0),0,0
100 COLOR ,15,(0,0)-(255,191)
110 RT=1.15
120 LINE (30,0)-(225,191),2,BF
130 POSITION (128,96),0,1
140 BCIRCLE(0,0),74,15,R1,,,BF
150 FOR I=1 TO 12
160 TH=RAD((3-I)*30)
170 Y=67*SIN(TH)*R1+4
180 X=67*COS(TH)-7
190 IF I>=10 THEN X=X-3
200 COLOR 2
210 CURSOR X+6,Y :PRINTI;
220 NEXT
230 L1=43 : L2=R1*L1
240 L3=55 : L4=RT*L3
250 L5=55 : L6=R1*L5
260 H=VAL(LEFT$(TIME$,2))
270 M=VAL(MID$(TIME$,4,2))
280 HR=RAD(30*H+M/2)
290 HX=L1*SIN(HR)
300 HY=L2*COS(HR)
310 H1=H
320 MR=RAD( 6*M)
330 MX=L3*SIN(MR)
340 MY=L4*COS(MR)
350 M1=M
360 REM
370 REM  -  --  --  -      ---
380 REM
390 S=VAL(RIGHT$(TIME$,2))
400 IF S=S1 THEN 390
410 T$=TIME$
420 H=VAL(LEFT$(T$,2))
430 M=VAL(MID$(T$,4,2))
440 S1=S
450 SR=RAD( 6*S)
460 SX=L5*SIN(SR)
470 SY=L6*COS(SR)
480 IF M=M1 THEN 580
490 MR=RAD( 6*M)
500 MX=L3*SIN(MR)
510 MY=L4*COS(MR)
520 M1=M
530 HR=RAD(30*H + M/2)
540 HX=L1*SIN(HR)
550 HY=L2*COS(HR)
560 BLINE(0,0)-(XM,YM),2
570 BLINE(0,0)-(XH,YH),2
580 BLINE(0,0)-(XS,YS),2
590 LINE(0,0)-(HX,HY),2
600 LINE(0,0)-(MX,MY),2
610 LINE(0,0)-(SX,SY),2
620 BLINE(-24,-87)-(23,-95),15,BF
630 CURSOR -24,-88:COLOR 1
640 PRINT T$;
650 GOSUB 720
660 XH=HX : XM=MX : XS=SX
670 YH=HY : YM=MY : YS=SY
680 GO TO 380
690 REM
700 REM --------------------------
710 REM
720 IF M 59 THEN 790
730 IF S 57 THEN 790
740 SOUND 1,440,15
750 FOR J=0 TO 10
760 NEXT
770 SOUND 0
780 MF=-1
790 IF MF=-1 THEN 840
800 IF(V<8)THEN BEEP 1 : BEEP 0
810 IF V<0 THEN 840
820 SOUND 1,880,V
830 V=V-2
840 IF M 1 THEN 880
850 IF S>1 THEN 880
860 SOUND 1,880,15
870 V=14 : MF=0
880 RETURN
890 REM--------------------------
900 F=220:SOUND 1,,15
910 FOR I=1 TO 4
920 C1=13*RND(1)+1:Y=RND(1)*40
930 D0=0:D=2*I
940 FOR X=-71 TO 245 STEP 4*I
950 D0=D0+D:Y=Y+D0
960 IF ABS(D0)>?*I THEN D=-D
970 SPRITE 1,(X,Y),I*4+12,C1
980 SOUND 1,RND(1)*F+F
990 NEXT:F=F*2
1000 NEXT
1010 BEEP0:RETURN
```

EASYCODE PART 1

A MACHINE CODE SIMULATION

Simon N. Goodwin

So that we can teach you the principles of machine code programming without worrying about what microprocessor you've actually got, this series uses a BASIC simulation. Clever, huh?

This series of four articles is aimed at anyone who would like to learn to program in machine code, the fastest and most intricate programming language on any home computer. Machine code is generally tens of times faster than BASIC; it is the language used for most sophisticated games and business programs.

This series will explain the principles which underly machine code programming rather than the nitty-gritty details of how to program the Z6509A 11-bit NMOS CPU with on-chip TTL I/O! The examples can be run on **any** popular computer which supports BASIC and a TV display. eg. SEGA.

MEAN MACHINES

When you set out to learn machine code, two major problems are likely to stand in your way. The first pitfall for the budding wizard is the unfriendliness of most 'monitor' programs. Monitors are programs which let you write machine code, in much the same way as a word processor lets you write letters.

Most monitor programs expect the user to type cryptic one-letter commands or devious mnemonics. On a Spectrum or TRS-80 the unpronounceable word "LDIR" means 'copy memory'. "3D0G" means 'go to BASIC' to the Apple monitor. Similar examples abound.

Things become even worse when you try to test your program. In BASIC, if you mis-type a line number you are told something like "Line not found". Most monitors can't detect such a mistake — the machine code will zoom off to a non-existent line, usually with non-satisfactory results! From that point onwards the computer is out of your control.

Mistakes like this often cobble the entire contents of the computer's memory, forcing you to reset the machine and load your program all over again. The sheer speed of machine code makes it hard to diagnose the exact location and time of an error.

NEW IDEAS

The second problem to be faced by a would-be machine code programmer is the intricacy of the language. Machine code is different for every type of processor, but all of the cheap processors incorporate these principles:

- The idea that data and program are equivalent
- The idea of storing information on a 'stack'
- 'flags' and 'registers'
- 'addressing modes'
- Number representation (binary, decimal, hex)
- 'bit' manipulation

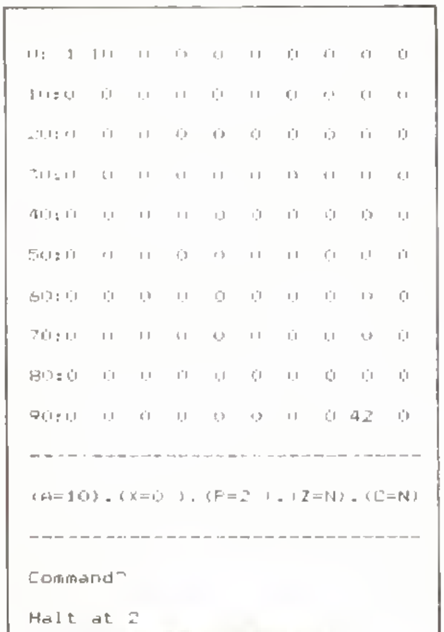
None of the principles are very complex, but most of them must be understood thoroughly before any useful programming can be done. To those six principles a seventh should perhaps be added:

- Jargon!

since, like other areas of programming, machine code has spawned a new vocabulary for humans as well as for computers. Jargon is, within limits, an efficient way of communicating ideas, so this series will not shy away from it. However, unlike the people you meet at computer shows, we will do our best to explain what we mean by each word before it is used!

THE UNIVERSAL CODER

This series features a standard BASIC program, developed and refined over three years, which demonstrates the first four'points. Once these are understood it is easy to see the relevance of the others. The demonstration should give you the confidence to move from this 'model' machine code to the real thing. At the end of the series we will duscuss the differences between the demonstration and real machine code.



```
0:  1  10  0   0   0   0   0   0   0   0
10:0   0   0   0   0   0   0   0   0   0
20:0   0   0   0   0   0   0   0   0   0
30:0   0   0   0   0   0   0   0   0   0
40:0   0   0   0   0   0   0   0   0   0
50:0   0   0   0   0   0   0   0   0   0
60:0   0   0   0   0   0   0   0   0   0
70:0   0   0   0   0   0   0   0   0   0
80:0   0   0   0   0   0   0   0   0   0
90:0   0   0   0   0   0   0   0  42   0
--------------------------------
(A=10).(X=0 ).(P=2 ).(Z=N).(C=N)
--------------------------------
Command?
Halt at 2
```

Fig. 1 The Easycode display.

Program 1 is the 'toolkit' which you will use to teach yourself machine code. You probably remember how BASIC didn't really make sense until you got hold of a computer and actually used the language. 'Easycode' (Program 1) lets you learn machine code the same way, by combining the essence of machine code with the messages and safety-checks of BASIC.

Easycode is presented in two parts. The first part, listed this month, is a complete toolkit which allows you to program in simple 'machine code'. The second part will add extra facilities, allowing you to experiment with 'assemblers', 'disassemblers' and 'stack operations'. These terms will be explained, by example, in Parts 3 and 4 of the series.

Program 1 provides all the facilities you need to enter, modify and test machine code programs. You can also store your work on tape for retrieval later. The next listing will consist of lines to be added to the first program — it won't be a program in itself.

EASYCODE

There are two places where information can be stored in a computer — the memory and the processor. Easycode lets you watch information (programs and data) being copied and manipulated inside the computer. The program is a kind of 'computer simulator'. A typical display is shown in Fig 1.

Easycode simulates a computer with 100 'memory locations', numbered from 0 to 99. A 'memory location' is a storage space within a computer — a kind of electronic pigeon-hole in which a single value can be stored. Sometimes a memory location is referred to as an 'address' — the name of a place within a computer.

Easycode shows the contents of a memory location as a whole number which may range between 0 and 99. That number might represent a letter of the alphabet, or a colour, or anything you like. In Fig. 1 you can see that location 1 contains the value 10. Location 98 holds the value 42. (Every computer should contain 42 somewhere!). Most of the memory contains the value 0.

The top 10 rows of the display show the contents of the computer's memory, from location 0 in the top left corner to location 99 at the end of the tenth line. The leftmost column is an index. All of the values in the other columns can be altered — they always show the value in the appropriate memory location. The effect is rather like having an integrated circuit with a glass top — you can read the computer's memory.

The twelfth line of the display shows the contents of the processor. Broadly speaking a processor does four things:

- It fetches values from memory
- It alters values once they are fetched
- It stores values in memory
- It changes the sequence of operations performed depending upon the values it contains

The processor must have some memory of its own, so that it can remember values once it has fetched them. Memory locations inside a microprocessor are called 'registers'. There are usually between two and 20 useful registers in a processor. Easycode has three registers, named A, X and P (registers often have one or two-letter names). The value stored in each register is shown, next to its name, on the twelfth line of the display.

The 'A' register, or 'Accumulator', is used to hold the temporary results of calculations. The 'X' or 'Index' register contains either results or the 'index number' (address) of a memory location. Index registers are used by a computer to 'mark its place' in data.

The 'P' register is the 'Program Counter'. Every microprocessor has a program counter. It contains the address of the 'instruction' which is being executed.

INSTRUCTIONS EXPLAINED

A computer decides what operation to carry out by examining values in memory. These values are called 'instructions' — different values cause different operations to be performed. One value might mean stop, another might mean 'JUMP' (change the value of the program counter) and so on. A machine code 'program' is just a sequence of instructions.

Instructions are numbers fetched when the program counter indicates

CLEAR	Blanks Easycode's memory and registers
HELP	Shows a summary of the commands
LOAD	Read memory contents from tape (or disk)
QUIT	Stops the Easycode BASIC program
RUN	Starts a machine code program
SAVE	Store memory contents on tape (or disk)
STORE	Changes the value at any memory location

Table 1. Easycode commands.

them. Values fetched for any other reason are 'data'. There's no reason why some locations shouldn't be both data and instructions at different times. The computer can produce data and then JUMP to it and treat it as instructions. This is potentially a very useful trick, which we will explore later in this senes.

The program counter determines which location is examined for the next instruction. Normally the computer steps from low-numbered locations to higher ones, just as BASIC goes from one line-number to the next. Some values may cause the computer to skip locations or go back to a lower-numbered location — these machine code instructions correspond to the 'GOTO' command of BASIC.

The last important components of a processor are the 'flags'. These operate rather like railway points, telling the computer whether it should go ahead (to the following instruction) or turn elsewhere. Easycode has two flags, labelled 'Z' for 'Zero' and 'C' for 'Carry'. Each flag may have two values — 'set' or 'reset'. Some computer makers take a high moral tone and label these values TRUE and FALSE respectively. Easycode shows a 'Y' next to a flag's name when it is set, and an 'N' when it is reset. The significance of these flags will become clear later.

PLEA AND JUDGEMENT

The last two lines of the Easycode display are used for commands and messages. Reports from the computer appear on the bottom line.

Commands are typed in capital letters on the line above. Table 1 is a complete list of the commands recognised by Program 1.

Since instructions and data are stored identically, the STORE command can be used to enter any kind of information. If you type STORE while the prompt 'Command?' is displayed the computer responds 'Address?'. 56A Stalingrad Mansions won't do — you must type the number of the memory location you wish to alter. If the value you type is not in the range 0 to 99 you are asked for another command.

Assuming that you typed a valid address, Easycode asks you for the value to be stored at that address. Enter another value between 0 and 99. Easycode takes it and stores it in memory. If you watch the display you will see the value appear.

Next you are asked for a value to be stored in the subsequent location. The sequence continues until the end of memory is reached or you type an invalid number (such as 100). You are then asked for a different address. Either select a new address and enter values as before or type 100 to halt the store operation and return to the 'Command?' prompt.

A SIMPLE PROGRAM

The simplest BASIC program is:
1 STOP
To write this in Easycode all we need to know is the instruction value which will cause a program to halt. Table 2 contains a full list of Easycode instructions. At the head of the list is 0 HALT — when the computer encounters a '0' instruction it will stop and display the value of the program counter.

In case of accidents, all of the computer's memory is filled with 0's when Easycode is first run. Wherever we start our program it will encounter a HALT and stop immediately. To confirm this, type RUN and then enter any address. Easycode loads the number you type into the 'P' register and then executes the instruction there. Notice that each memory location is flashed as Easycode reads an instruction from it.

We'll try something slightly more complicated next. The next program contains two instructions (wow). The first instruction loads a value into the 'A' register and the second one is the HALT which we have came to know and love.

Code	Mnemonic	Purpose
0	HALT	Stop machine code program.
1	LOAD A;n	Put next memory contents into A.
2	LOAD A;@n	Put contents of address n in A.
3	STORE A;@n	Put contents of A at address n.
4	LOAD A;X	Copy contents of X into A as well.
5	ADD A;n	Add next memory contents to A.
6	SUB A;n	Subtract next memory contents from A.
7	SUB A;@X	Subtract the contents of the address numbered in X from the contents of A.
8	JUMPNC;n	Go to address n if carry is not set.
9	JUMPNZ;n	Go to address n if zero is not set.
10	JUMP;n	Go to address n.
11	LOAD X;n	Put next memory contents into X.
12	LOAD X:@n	Put contents of address n in X.
13	STORE X;@n	Put contents of X at address n.
14	LOAD X;A	Copy contents of A into X as well.
15	ADD X;n	Add next memory contents to X.
16	SUB X;n	Subtract next memory contents from X.
17	LOAD A;@X	Put the contents of the address numbered in X in the A register.
18	STORE A;@X	Put number in A at the address in X.
19	ADD A;@X	Add the number at address X to A.

'n' represents any value between 0 and 99.

Table 2. Easycode Instructions (8K version).

If you consult Table 2 you will see that instruction value 1 means 'LOAD A;n'. This instruction takes up two memory locations (HALT only took one). The first location contains the instruction (1). The following location contains the data to be loaded, so that the sequence of values 1 2 will cause the value 2 to be loaded into the A register. Use the STORE command to put the values 1 2 0 into memory from location 0 onwards.

When you RUN from location 0 you will see the 1 flash, then the value 2 will appear in the A register. The computer skips over location 1 (because it is data — part of the LOAD instruction) and flashes the contents of location 2 — the HALT.

ASSEMBLER MNEMONICS

It may seem rather pointless to use these odd names: 'HALT', 'LOAD A;2' and so on when we can't type them into the computer — we have to use the numeric values from Table 2 instead. These names are called 'mnemonics', (pronounced nem-on-iks) which is Greek for 'reminders', and they're designed as an aide memoire for programmers. The idea is that a sequence such as:

```
0: LOAD A;1
2: ADD;1
4: JUMP;2
```

makes a little more sense than the string of digits 1 1 5 1 10 2! Most machine codes use mnemonics, although they vary in detail from one processor to another. Easycode has 20 instructions (more will be added later) and consequently 20 mnemonics, listed in Table 2. Each mnemonic has a name (eg HALT, ADD, LOAD) and most of them have 'arguments' too — these describe what information is used and where it is stored. The instruction LOAD A;1 corresponds to A=1 in basic. ADD A;1 corresponds to A=A+1. JUMP;2 is similar in effect to GOTO 2.

The format of Easycode mnemonics is very similar to that of real machine code, although a semi-colon is used as a separator rather than a comma since BASIC INPUT statements tend to do strange things with commas!

Enter the sequence 1 1 5 1 10 2 into memory from location 0 onwards. When you RUN the program (starting at 0 once again) you will see Easycode counting in the A register. Watch the display as Easycode counts. Locations 2 and 4 flash alternately as the instructions within are executed. The value in the program counter P changes back and forth, and the accumulator A counts up steadily. You can pause the program at any point by pressing the SPACE key. Type an end of line to stop the program or any other key to re-start it. You can use the end of line key to halt the program immediately if you wish.

USING THE FLAGS

If you let the count continue all the way up to 99 you will see something interesting happen. When A contains 99 and 1 is added there isn't room for the value 100. Eosycode, like all machine codes, simply throws away the extra digit — the one — and counts from 0 again. The computer has, in effect, said '99 plus 1 is 0, carry 1'. When the value in A 'overflows' the carry flag becomes set — the display shows 'C=Y'.

The carry flag is set whenever an operation results in a carry or a borrow. When you try to SUBtract 1 from 0 you will get 99 borrow 1 — the register will hold 99 and, once again, the carry flag will be set.

The zero flag works in a similar way, but it becomes set whenever an operation ends up with a zero value. The zero flag also becomes set if you LOAD or STORE a zero. You can use this rule to test for any value — just load the number to be tested into the A register and subtroct the value you want to test for. If the zero flag is set after that, you know that the number and the expected value were the same.

This flag-waving is all very well, but it seems rather pointless unless we can tell the computer to make decisions depending on the value of the flags. There are two Easycode instructions which test the flags, doing different things depending upon what they find. The instruction JUMPNZ;n tells the computer to JUMP to the instruction at location 'n' if the zero flag is NOT set. If the flag **is** set, the computer simply skips over the JUMPNZ and performs the subseguent instruction. The JUMPNC;n instruction is identical except it tests the carry flag, producing the effect of the BASIC line:

```
IF C < > Y THEN GOTO n
```

It is easy to see how we can use this instruction. Change the contents of locations 3 and 4 to 10 and 8 respectively. Now our program is:

```
0: LOAD A;1
2: ADD A;10
4: JUMPNC;2
```

The program now counts guickly until it tries to add 10 to 91 — the result is 1, carry 1, and the program 'falls through' to location 6.

One important thing to note about machine code is that the computer can't tell instructions and dato apart. This can have unfortunate consequences if you jump to the wrong address. Consider what would happen if we started the above program at address 1 instead af 0 . . .

The computer finds a 1 at address 1. It treats that as LOAD A;next, and puts the value 5 (the ADD instruction!) into A. Next it finds the 10 at address 3. 10 means JUMP, so it jumps to the address in location 4 — an 8. Notice that we've ended up with a completely different program, simply by starting one location later.

Sometimes mistakes like this will cause the computer to try to execute a non-existent instruction — a value greater than 19, for instance. A real computer might do unpredictable things in such a circumstance, but

Easycade can detect the error. If you make that kind of mistake Easycode stops and prints the message 'Unknown Instruction'.

THE PROGRAM

Program 1 is a complete listing of the Easycode program for the TRS-80 Model 1 or Video Genie. The only requirements are a display at least 32 columns wide and 16 lines long, 8K of user memory, string handling, and a BASIC which allows characters to be read from the keyboard as a program runs. The expanded version of the program requires a 40 by 16 display (or larger) and 16K or memory.

The listing is extensively commented, so that it should be possible to work out the effect of instructions from the listing even if you can't make it out by experimentation. Same parts of the program have been deliberately kept simple rather than efficient, on the grounds that it is better to have a slow correct program than a speedy one which doesn't always work!

Once you have converted the program it should be easy to identify the parts which can be accelerated. Keep to the same line numbers as much as possible, since this will make it easier to add the extra instructions introduced in Part 2. On a Spectrum or ZX81 you should divide all of the line-numbers by five.

Next month we'll explain how to convert the program for almost every machine under the sun, and we'll demonstrate multiple precision arithmetic, input-output and even moving graphics. Don't miss it!

EASYCODE PART 2

Simon N. Goodwin

Now that we've presented the Easycode program for one machine and explained the basic principles, here's how to convert it for a variety of machines. We also have the first of our example routines.

In last month's article we introduced 'Easycode', a program which lets you learn machine code (almost!) as painlessly as if it were BASIC. If you missed that article you need a copy of the listing from a back-issue of the magazine. This month we explain how you can convert the program for other machines. We also press on with example programs, including arithmetic routines and even memory-mapped graphics!

EXAMPLE PROGRAMS

Last month we experimented with some trivial Easycode programs. This issue we use most of the instructions, and explain how to crack one of the major problems of machine code programming — the storage of numbers larger than the computer can fit in a single location.

Program 1 performs a common task — it adds up the values in locations 50-59, and stores the total in location 99 (at the end of memory). We've changed the format of our Easycode listing so that it closely resembles the format of a 'real' machine-code or assembler listing. The program is divided into four vertical columns. The first column (0, 2, 4, 5 etc) is the 'address' at which the instructions are stored. Program 1 occupies 16 locations, since it starts at address 0 and goes on to 15 (the JUMP at address 14 uses two locations).

TABLE 1

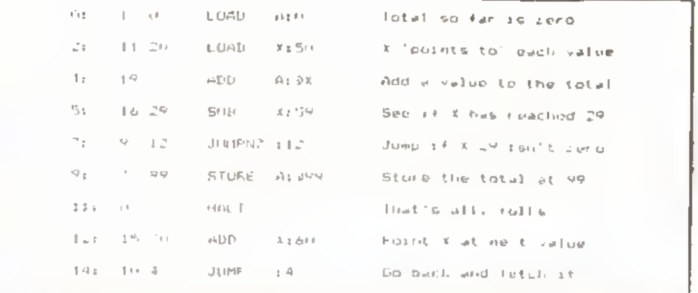
1000	CLEAR command	11000	RUN command
3500	Scan keys	11500	Fetch instruction
4000	Clear line	12000	SAVE command
4500	Accept number	12500	Accept file name
5000	Update registers	13000	LOAD command
6000	Position cursor	14000	HELP command
8000	Display number	15000	QUIT command
9000	Draw full screen	16000	STORE command
10000	Accept command	16500	Update memory
10500	STOPPED message	20000	Execute instruction

Easycode routines.

The second column contains the data which should be stored in each location. The instruction code to load the 'A' register is 1, hence location 0 contains 1. The value to be loaded is 0, so location 1 contains 0 — and so on. At the end of the listing, the code 10 corresponds to a JUMP instruction. The 4 following it (in location 15) tells the computer where to JUMP to. The numbers in the second column (1, 0, 11, 20 etc) are the ones you enter using the STORE command.

Column three contains the mnemonic form of the instructions. A computer can't understand mnemonics directly — they are just shorthand for human beings. You can get programs — known as assemblers — to convert mnemonics into the corresponding numbers. The expanded version of Easycode contains a built-in assembler, as well as a disassembler which performs the reverse function, taking the numbers stored in memory and converting them into their mnemonic equivalents. The program lines needed to add these features to Easycode will be published in next month's CT.

The last column in the listing contains 'comments'. These are English phrases added by the programmer to make the purpose of the code more clear to someone reading the listing. Comments are similar to REMS in BASIC; they are ignored by the computer.



```
0:   1  0   LOAD    A:0     Total so far is zero
2:  11 50   LOAD    X:50    X 'points to' each value
4:  19      ADD     A:@X    Add a value to the total
5:  16 59   SUB     X:59    See if X has reached 59
7:   9 12   JUMPNZ  :12     Jump if X-59 isn't zero
9:   [illegible] 99   STORE   A:99    Store the total at 99
11:  0      HALT            That's all, folks
12: 15 60   ADD     X:60    Point X at next value
14: 10  4   JUMP    :4      Go back and fetch it
```

Program 1. Adding up a list.

ADDING FOR BEGINNERS

The first two lines of Program 1 are straightforward. STORE the program and RUN it from address 0. You see the value 0 loaded into A, and 50 loaded into X. Both registers are being used for their designed purpose — the A register contains the running total and the X register is used as an index, 'pointing to' the address of items in the list. This is where the instructions using '@X' come in useful. They fetch, add or otherwise manipulate the contents of the memory location which is numbered in the X register.

If X contains 3, LOAD A:@X will fetch the contents of location 3. The instruction would be read 'Load A with the value at location X'. The third instruction adds the contents of the location pointed to by X. The X register is being used rather like a bookmark, keeping track of the place the computer has reached.

It's all very well having the X register marching through memory, pointing accusingly at everything it passes, but how do we tell when we have reached the end of the list? We know that the last item is at address 59, so we must stop once X reaches 59. Most processors have an instruction which 'compares' a register with a value. These instructions work by subtracting the required value and then setting the flags according to the result. In Easycode we have to use an explicit subtraction (wiping out the previous result in the process), but we shall see that this is not a problem.

TABLE 2

I,J,K	Miscellaneous integer counters etc.
ROW	Cursor vertical address 1 (top) - 16.
COLUMN	Cursor horizontal address 1-32
T$,T1$,T2$	General purpose strings
P	Program counter
ABRT	Non zero when program has been aborted
CARRY	Carry flag
ZERO	Zero flag
R(0)	Accumulator (register 0).
R(1)	Index register (register 1)
M(0) M(99)	Memory array

Variable list

To test for the value 59 we merely take 59 away from the contents of X, with a SUB X;59 instruction. If the result is not zero, X does not contain 59; we must go on to the instruction at location 12. If the result IS zero, Easycode ignores the JUMPNZ (jump if not zero) at location 7, and continues to the next instruction, at location 9. This stores the value in A 'at' location 99. The program halts when the '0' code at location 11 is encountered.

Meanwhile, if we're still trolling along the list, we've arrived at location 12 with a problem on our hands. Depending upon your

CONVERTING EASYCODE

The program was deliberately written for easy conversion to run on other computers. As much as possible, machine-dependent routines to read the keyboard, position the cursor and so on have been collected in one place. 'Easycode' uses a subset of the BASIC language — it will run on almost any computer with BASIC, a TV display and the facility to handle strings of characters.

Table 1 lists the routines which make up the Easycode interpreter. More than half of the routines should run without changes on just about any BASIC computer. In this section of the article we will go through the routines one by one, explaining their purpose and giving examples of the conversions required. None of the REMS (which include comments introduced by an apostrophe) need be entered.

There are five points which should be observed throughout. The listing assumes that the program is running on a computer which handles floating-point (decimal) numbers (a TRS-80 or Genie). A few systems only recognise integers (whole numbers). The program will work on such machines, so long as they have the other required features, but INT expressions, used to round-down numbers, should be ignored; integer BASIC always rounds down. Lines marked 'F.P. BASIC only' may be omitted on integer-only systems.

The CLS command is used to clear the screen at various points. If your computer doesn't have a clear screen command, you can probably simulate it by printing a special 'clear' character or group of characters. Alternatively you can set up a 'CLS' subroutine which calls the 'clear line' routine (line 4000) for every line of the display.

The next three 'general' points are addressed mainly at users of Sinclair computers, although they may concern a few other users. The program uses the format IF expression THEN line number. This may have to be entered in the form IF expression THEN GOTO line number.

Easycode assumes that array subscripts start at zero. If the instruction PRINT R(0) gives an error message, you'll need to add one to the subscript of every array reference in the program. In retrospect, this stems from bad design — in the interests of portability the original program should have ignored the existence of the zero subscript.

Table 2 contains a list of the 14 variable names used. These all differ in the first two characters, and they do not contain BASIC words (hence ABRT not ABorT). If your computer doesn't allow strings to have names or more than one character you must rename T1$ and T2$. If necessary, enter a LET statement at the start of every line which presently begins with a variable-name.

A MODEL PROGRAM

The first few lines of the program are used to set up an 'empty' computer model. They are consequently used whenever the program is first RUN, and after a CLEAR command, which works by starting the program again from scratch. Line 1000 will not be needed on most computers — it sets all variables to zero and reserves space for up to 100 characters of strings. The DIM statements in line 1010 must be altered, as described earlier, if your computer does not allow zero subscripts.

The SCAN KEYS subroutine will run exactly as listed on a Spectrum or Dragon. The INKEY$ function returns an empty string ("") if no key has been pressed; otherwise it returns the character corresponding to the key. Note that the end-of-line key is assumed to produce CHR$(13), and the space key is expected to give CHR$(32). The key feature (sorry) of this routine is that it should go on without waiting if no key is pressed. If you don't know how to do this on your computer, consider using joystick control (usually by a PEEK) instead of the keyboard.

The CLEAR LINE subroutine should not need changing. It simply positions the cursor, prints 32 spaces, and puts the cursor back at the start.

ACCEPT NUMBER, from line 4500 onwards assumes your computer uses the ASCII character set. If necessary replace the colon in line 4510 with the character which follows "9" in the sequence recognised by your machine. The action of line 4510 is to reject any entries which do not start with a digit. A simple greater-than-or-equal-to-"9" test is unsatisfactory since, by convention, the string "9" is greater than "9"

The UPDATE REGISTERS subroutine is straightforward.

POSITION CURSOR tells the computer that the nect character to be printed should appear on row ROW and column COLUMN, assuming that the top left position on the display is ROW 1, COLUMN 1. On a Dragon use PRINT @ COLUMN+ROW*32−33, "";. A BBC Micro will require PRINT TAB(COLUMN−1, ROW−1); The Spectrum version is PRINT AT ROW−1,COLUMN+1;. Atari owners should use LOCATE COLUMN−1, ROW−1. if you've got a terminal which recognises the VT52 escape sequences, PRINT CHR$(27);"Y";CHR$(31+ROW), CHR$(31+COLUMN); should work. if the worst comes to the worst you may have to HOME the cursor to the top left corner and print 'down' and 'right' characters repeatedly.

Next we come to the DISPLAY NUMBER subroutine, which starts at line 8000. This prints exactly two characters which indicate the value of N, from 0 to 99. The only tricky thing here is making sure you output two characters (one of them a space) for values less than 10. If your computer allows PRINT USING, use it!

The DRAW FULL SCREEN routine contains only one potential pitfall — the multiple NEXT statement in line 9110. Some computers will require separate statements for each loop (NEXT J amd NEXT ROW).

The subsequent three routines, ACCEPT COMMAND, STOPPED and RUN COMMAND, should not need changing. FETCH INSTRUCTION contains a single odd statement — the ON . . . GOTO at line 11680. Space has been left for this to be written out in full:

```
IF I=1 THEN GOTO 20000
IF I=2 THEN GOTO 20100 etc.
```

But a computed GOTO would work as well:

```
GOTO 19900+(100★I+400★(I>16)
```

so long as an expression such as 2>1 prints as '1' on your system (if it returns '−1', replace the second plus in the computed GOTO with a minus). You may have to use a mixed approach if your BASIC won't allow long lines.

The SAVE and LOAD subroutines occupy lines 12000-13240. These are perhaps the most difficult part of the program to convert successfully. If in doubt, fix these last. The routines merely SAVE and LOAD the array M() under the name in T$. Most computers require that you OPEN and CLOSE the data file before and after manipulations.

On a Spectrum you can get by with just SAVE T$ DATA M() and LOAD T$ DATA M(). Put ★"m",1; before T$ to save and load to microdrive (through cassette is almost as quick!). Most other computers use PRINT and INPUT to access files. The only difference from normal (console) access is a 'channel number' which tells the computer to use the cassette or disc, rather than the display. On the Genie '−1' is the channel number and '£' (typed as a hash) identifies it.

The rest of the program should be easy to convert, since it uses very simple statements. If your computer doesn't recognise PRINT, IF, GOTO, GOSUB and assignments, you're in real trouble!

NOTE: The programs at the end have been converted for the SEGA.

approach to arithmetic, you'll find the next explanation very simple or very devious — please bear with us in either case.

When we reach address 12, register X contains 59 less than the address of the place where we got our last value from; or 60 less than the address of the value we need next. At first sight this is a hard problem to sort out, because we must have taken 59 away from X when there was less than 59 in X to start with! We can't store negative numbers in Easycode, so what's the result?

You can guess the answer if you remember the way the Carry flag works. If the number goes over 99, Easycode sets the Carry flag — carrying 100, if you like — and leaves the remainder in the register. Hence 69 + 42 gives 11, carry one. So what is 50 — 59? The answer, according to Easycode and (in principle) every other micro machine code, is 91, borrow one. The sum is treated as 150 — 59. This may bring back memories of school long-division — if it does, sorry!

This rather arithmetical explanation has a point (in case you were wondering!). Since 'carry' and 'borrow' work exactly the the same way in machine-code, it follows that, whatever number you start with, you'll get the same number back if you subtract, and then add, any other number. This is obvious in normal arithmetic, where negative numbers are allowed, but it seems odd when you can say:

50 — 59 = 91
91 + 60 = 51

The ADD on line 12 has the effect of setting X to the value it had before the subtract, plus one. So we didn't lose the result during the subtract, after all.

Now X is pointing to the next value to be added to the total. We can go back to the ADD A;@X instruction, using a simple JUMP;4.

CHEQUERED FLAGS

STORE and RUN the program until you're happy that it works. Put different values in locations 50 to 59 to test it. Hopefully you're not flagging (sorry) yet, because there's another problem to be solved. What happens if the total is more than 99?

We can't store such a total in the A register, or in any one memory location, since there's only room for values between 0 and 99. But what, I hear you whistling in the dark, is to stop us using two memory locations? After all, we can only represent values between 0 and 9 with a single digit, but that doesn't stop us sticking them together in clumps to make tax demands and such-like.

We can count up to 9999 if we use two locations, one for the 'hundreds' and one for the 'units'. Since 10 individual locations can only contain separate numbers that add up to a maximum of 990, two locations will be plenty. Program 2 solves the problem. Location 98 contains the hundreds and location 99 the units of the result. While the program runs, the units are stored in the A register. STORE and test this program too.

```
0:   1  0  LOAD   A;0       Clear the total
2:   3 98  STORE  A;@98     No hundreds yet
4:  11 50  LOAD   X;50      X points to the list
6:  19     ADD    A;@X      Add an item
7:   8 19  JUMPNC ;19       Has it overflowed
9:   3 99  STORE  A;@99     Yes - store it and
11:  2 98  LOAD   A;@98     Fetch the hundreds
13:  5  1  ADD    A;1       Add one hundred
15:  3 98  STORE  A;@98     Put the new total back
17:  2 99  LOAD   A;@99     Retrieve the units
19: 16 59  SUB    X;59      Is X 59 zero
21:  9 26  JUMPNZ ;26       Jump on if not
23:  3 99  STORE  A;@99     All done, store units
25:  0     HALT             Finished!
26: 15 60  ADD    X;60      Advance X by 60-59
28: 10  6  JUMP   ;6        Round again!
```

Program 2. A better adder.

```
0:  11 90  LOAD   X;90      Point to bottom line
2:   1  1  LOAD   A;1       1 is data to be moved
4:  18     STORE  A;@X      Position the '1'
5:   4     LOAD   A;X       Copy the position into A
6:   6 99  SUB    A;99      Have we reached the end
8:   9 13  JUMPNZ ;13       Not yet
10: 18     STORE  A;@X      Economically clear the data
11: 10  0  JUMP   ;0        Start from scratch
13:  1  0  LOAD   A;0       Draw a '0' after the '1'
15: 18     STORE  A;@X      Position the '0'
16: 15  1  ADD    X;1       Advance to the next address
18: 10  2  JUMP   ;2        Loop round to put '1' there
```

Program 3. Moving graphics.

TRANSPUT

Our final programming topic this month concerns what is known as 'memory-mapped I/O'. I/O stands for Input/Output, alias communication between the computer and someone or something outside. Followers of the language Algol 68 tried to replace this rather ugly term with the invented word 'transput', but, sadly, it didn't catch on (rather like Algol 68!).

One of the biggest problems for the machine-code programmer is I/O — you can't just say INPUT or PRINT and watch words magically appear on the screen. Most modern computers use memory-mapped I/O, which means that they communicate with the outside world just as they do with their own memory — by storing and retrieving information at certain locations.

The electronics to drive the video display of most micros is quite simple. In effect, a set of memory addresses are connected simultaneously to the computer (which can read and write to the addresses) and to electronics which drives the TV or monitor. The electronics scans through the memory 50 times a second, producing a picture signal for the display. The display is produced by turning a 'dot' of light on and off as it scans across the picture, so that if the dot is on the screen glows and if it is off the screen is dark.

Imagine that the processor stores a selection of numbers in the first half of the display memory, and zeros in the second half. Depending upon the exact electronics used, this will produce a blank screen at the bottom and a jumble of dots or characters at the top. From this you can see that the more memory you allocate to the display, the more dots you will be able to control, and hence the higher the resolution of the display. So far we've assumed a 'Yes/No' value for each dot. If you use still more memory you can add intermediate values to give the effect of

The Easycode simulator takes memory mapping to its logical conclusion — all of the memory is displayed, all of the time. To move a dot across a computer screen you move a value through the screen memory, wiping the old position before each move. We can produce moving graphics in Easycode (slowly), by moving a value through memory in exactly the same way. Program 3 performs this task, moving the value '1' along the bottom of the screen.

STORE Program 3 and play with it. Slow though it is, it demonstrates exactly the technique used in your favourite arcade games. Of course, most shapes are made up of more than one point, but it is easy to see how a group of points could be made to move together. When you get bored with the horizontal movement, change the value at address 1 to 22, and store 11 at address 17. The graphics should now move diagonally. See if you can work out how to make them move up instead of down.

END OF PART 2

This month we've shown how the Easycode instructions work. If you intend to learn the principles of machine code, it is important that you experiment with the instruction set. Why not write a program to multiply the value in the A register by the value in X? You'll need two locations for the result. As an experiment in indexing, write a program to count all the occurences of a specific value (say, 0 or 1) in Easycode memory. Example solutions will be presented in the next issue of CT.

EASYCODE PART 3

Simon N. Goodwin

In this article of our Easycode series we let you talk to your computer in English (almost!) and show you how to have stacks of fun.

This month we'll add a number of extra features to the Easycode language. The new features make Easycode even more like 'real' machine-code — in fact the only major difference left will be the lack of binary operations, which we will explain next month.

The new program lines also give you a small 'assembler' and 'disassembler', via the added commands 'ASM' and 'DIS'. These facilities are not as versatile as the real thing, but they're still useful. They're also easy to use 'interactively' — you can chop and change your program clearly and fast.

The lines in Listing 1 are not a complete program — they are EXTRA lines to be added to those published in the March 1984 CT. This means that we can print a program which is longer than we could otherwise, but it isn't very helpful for intermittent readers! Back issues can be obtained using the coupon on Page 76.

CONVERTING THE PROGRAM

The extra lines restrict the range of computers on which the program can run, since they require a few extra features. String arrays are needed to run the assembler and disassembler. The stack display requires a screen 40 columns wide, although it could be programmed horizontally if more than 16 lines are available. The listing is for the TRS-80 Model 1 or Video Genie. The complete program requires 16K of BASIC memory.

Most of the program is fairly easy to convert, bearing in mind those restrictions and the notes published last month. You may need to alter the DIM statements on line 1030 to dimension each variable separately and to tell the computer how long each string may be (eg DIM D$(MAX,10)).

Microsoft BASIC string handling is used, as on Apple, Commodore and recent Acorn machines. Line 18520 compares D$(I) with the leftmost characters of T$ (making sure that the number of characters is the same in each). If your BASIC uses fixed-length strings you will need to take trailing spaces into account. The other tricky line is 18535, which sets T1$ to contain all the characters in T$ after position 'I'.

The '+' operator is used to stick strings together, so that PRINT "Easy"+"code" gives:

Easycode

This will work on most computers other than Ataris, where devious use of string-slicing will be needed. String '+' is only used in Easycode to make error messages, so it should be possible to make the program work without the feature — so long as you find another way of reporting problems.

The DATA in lines 30000 onwards is copied into the arrays D$() and E(). If your computer won't allow READ and DATA statements, you must use 52 assignments instead (my sympathies).

Listing 1 has been automatically mixed back into the original program. The combination worked at once, so the listings should be compatible unless our layout department have scalpel problems! If you type in the lines and they won't work, please check that you've made compatible changes in both parts of the listing, and you haven't mistyped any line-numbers.

USING THE ASSEMBLER

The assembler converts mnemonics like 'LOAD' and 'HALT' into the numbers recognised by the computer. You type in relatively-readable lines such as 'ADD A,7' and the routine generates the values '5' and '7'. The numbers are put into memory (you can see them appear, just as with the 'STORE' command) and you can type another line.

The main difference between the ASM facility and the assembler of a real computer is the lack of 'names' for locations and values. A real assembler would allow you to give a location a name, such as 'TOTAL', and then refer to it by name: STORE A,@TOTAL. This feature is missing from ASM as it would make the routine slower, less portable and harder to use (you'd have to check for names used before they were identified, for instance). Since Easycode only uses 100 memory locations that lack of naming is not a major drawback.

To use the assembler, type ASM in response to the Command prompt. You will be asked where you want to store code, just as with the STORE command. Enter a sensible address or '100', if you want to chicken out. As you type the mnemonic lines, one by one, the computer stores the appropriate codes. To leave the assembler type '100' instead of a mnemonic.

There are four possible error messages when you use the assembler. 'Unknown' appears if the computer can't recognise the mnemonic from its list of 26 possibilities. If you entered ADD B,1 you'd get the message, since there is no 'B' register. You'd also get it if you typed TWIDDLE THUMBS as there's no Easycode instruction to do that. The only special word you can use is HELP, which prints a list of the mnemonics allowed. If you have trouble, use the HELP command to see the format required — in particular, avoid using extra spaces.

The message 'Too long' appears if the computer understands the first part of an instruction but didn't expect the rest. Trailing spaces or words may cause that message. If you try to use a name instead of a number, or type a silly value, the computer comments 'Incorrect number'. Nothing is stored if you make a mistake. You are asked to type the line again.

There is one obscure error message. If you try to store an instruction which needs two locations at address 99, the computer displays 'Only one memory space left'.

THE DISASSEMBLER

The disassembler converts numbers back into mnemonics — the opposite of the assembler. Type the command DIS and then tell the computer where in memory you wish the disassembly to begin. Each location will be examined and the appropriate mnemonic printed. If the location contains a value which does not correspond to an instruction the value is assumed to be data, and the mnemonic 'Data code' appears.

The disassembler prints the value and name of each instruction. Each disassembly consists of 13 lines, after which you can opt to stop or continue. If you continue there is a deliberate small overlap of addresses. Experiment with the disassembler to see how it works. Try disassembling data as well as programs, and see what happens if you start the disassembly in the middle of a program or instruction. This shows how the computer can produce strange results if programs are RUN from odd places.

SAME OLD PROBLEMS

Last month we set a couple of problems for keen programmers. Listing 2 shows a solution to the second, and simpler, problem: counting the number of occurences of a given value in memory. Hopefully you didn't have much trouble solving this. The comments at the side of the listing should make it fairly easy to understand, especially as it is quite similar to the 'adder' program published last issue.

You might like to try out the assembler by entering the mnemonics of Listing 2. Disassemble it to make sure you've made no typing errors, and then RUN it from location 0. As written it searches locations 0 to 99 for the value 1. The total is stored (eventually!) at location 99.

```
0:   1  0    LOAD A;0       Count so far is zero
2:   3  99   STORE A;@99    Store total
4:   11 0    LOAD X;0       X is start of search
6:   17      LOAD A;@X      Fetch a value
7:   6  1    SUB A;1        Compare it with one
9:   9  17   JUMPNZ;17      Jump if it doesn't match
11:  2  99   LOAD A;@99     Fetch count so far
13:  5  1    ADD A;1        Add one to the count
15:  3  99   STORE A;@99    Store the new count
17:  15 1    ADD X;1        Point to the next location
19:  9  4    JUMPNZ;4       Repeat unless back at zero
21:  0       HALT           End of program
```

Listing 2. Searching for a value.

The other problem — multiply A by X — was rather more difficult especially since the 'real' machine code solution involves shifts and binary arithmetic, which are not available in Easycode. The problem was set so that we can demonstrate how the new instructions introduced this month are useful — next month we will compare the refined Easycode with 'real' solutions for the Z80, 6502 and 6809 microprocessors.

Listing 3 uses only the instructions explained so far to perform the multiplication. The result is stored in locations 98 and 99 (the biggest value, 99 x 99, is 9801) and locations 96, 97, 98 and 99 are used for temporary results.

The approach used by the program is quite simple. The value in the A register is added to itself repeatedly. Each time, one is subtracted from the value in the X register. When X reaches zero the multiplication is finished. The program consequently doesn't do multiplications involving 0 correctly, but that would be easy to fix

There are a number of more serious complications. The first is that we can't add the old value of A to the running total very easily without making the program alter itself (using ADD A;number and 'plugging' the number into the program). Such a solution is rather messy and prone to error. Terrible things happen if the wrong location is accidentally 'plugged'! Instead we store the number at location 96 and point X at it, using the ADD A;@X instruction. In turn, we then need to use location 97 to save the count which was in X

```
0:    1   2   LOAD A;2       First No. to be multiplied
2:    11  2   LOAD X;2       Second number (hence 2 x 2')
4:    13  98  STORE X;@98    Save X temporarily
6:    3   99  STORE A;@99    Save A too
8:    11  98  LOAD X;98      Point X at the old X value
10:   7       SUB A;@X       Compare the old X with A
11:   8   19  JUMPNC;19      Jump if A exceeds the old X
13:   2   98  LOAD A;@98     Put old X in A
15:   12  99  LOAD X;@99     Put old A in X (swap)
17:   10  23  JUMP;23        Carry on
19:   2   99  LOAD A;@99     Restore A
21:   12  98  LOAD X;@98     Restore X
                             Multiply starts here
23:   3   96  STORE A;@96    Save A, to be added repeatedly
25:   1   0   LOAD A;0       Fetch zero to...
27:   3   98  STORE A;@98    Clear the 'hundreds' total
29:   2   96  LOAD A;@96     Restore A
                             The 'adding loop' starts here
31:   16  1   SUB X;1        Count one less loop required
33:   9   38  JUMPNZ;38      Plug on unless we've reached 0
35:   3   99  STORE A;@99    Store the units
37:   0       HALT           That's all folks
38:   13  97  STORE X;@97    Save the count for later
40:   11  96  LOAD X;96      Point X at the multiplier
42:   19      ADD A;@X       Add it to the total yet again
43:   8   55  JUMPNC;55      Jump if it fitted in A
45:   3   99  STORE A;@99    It overflowed, save remainder
47:   2   98  LOAD A;@98     Fetch 'hundreds' so far
49:   5   1   ADD A;1        One more hundred
51:   3   98  STORE A;@98    Store the new 'hundreds' total
53:   2   99  LOAD A;@99     Fetch remainder (units & tens)
55:   12  97  LOAD X;@97     Get the count again
57:   10  31  JUMP;31        Do the next add if necessary
```

Listing 3. A crude multiplier.

Whenever we add a value to that in A we must check that the total wasn't more than 99, otherwise an overflow (carry) has occurred and we should add one to the 'hundreds' figure at location 98. Look at Listing 3 if this doesn't make sense.

There is one more potential problem. The program so far would calculate 3 x 9 far more slowly than 9 x 3, since the first sum would involve nine additions and the second only three (assuming the second figure was loaded into the X register). This snag is easily avoidable. The program automatically compares A and X at the start and makes sure that the lowest value is in the X register, by swapping A and X if need be.

RUNNING DRY

If this program looks rather daunting, don't worry . WE couldn't make it work for a while! Table 1 shows a good way of testing programs like this. The technique is called 'dry running', and it is rather like running the program on paper before you let the computer at it. Dry running is a very useful skill as it can be applied to almost all languages.

To dry run a program you start by heading a sheet of paper with the names of all the variables or locations which are altered by the program. In the example these are the registers A and X, plus locations 96 to 99 which are used by the 'multiply' program. Dry runs are best for testing programs with few variables — often true of machine code. You need one extra column to record the program line' — in BASIC this would be the line number, while in Easycode it is the value of the P register.

TABLE 1

P	A	X	96	97	98	99
0	2	–	–	–	–	–
2	–	2	–	–	–	–
4	–	–	–	–	2	–
6	–	–	–	–	–	2
8	–	98	–	–	–	–
10	0	–	–	–	–	–
19	2	–	–	–	–	–
21	–	2	–	–	–	–
23	–	–	2	–	–	–
25	0	–	–	–	–	–
27	–	–	–	–	0	–
29	2	–	–	–	–	–
31	–	1	–	–	–	–
38	–	–	–	1	–	–
40	–	96	–	–	–	–
42	4	–	–	–	–	–
55	–	1	–	–	–	–
31	–	0	–	–	–	–
35	–	–	–	–	–	4
halt	4	0	2	1	0	4

Work your way through the program, using a new line every time a value is stored. Write down the value of P as you go along, as a 'key' The first line stores 2 in register A, hence the '2' in the 'A' column. The next line stores '2' in register X At any time each current value can be seen by consulting the lowest entry in the column concerned (if there is no entry the value is unknown).

This technique isn't a good way to test a large program (you'd need a sharp pencil and very big pieces of paper) but it is ideal for testing short complicated sections. Dry run Listing 3 (you'll need 20 lines) and compare your results with Table 1.

THE JOY OF STACKS

If you've typed in the new lines you will have spotted the extra column and register on the display by now! The column is the 'stack' and the new register 'S' is called the 'stack pointer'. The stack is an area of memory inside the computer (it could use some of the 100 locations but we chose not to, to avoid confusion). Unlike most memory, you can't put values into the stack willy-nilly — you have to use special instructions.

The stack works — as you might expect from its name — rather like a pile of values. You can 'PUSH' numbers onto the 'top' of the stack or 'POP' them off the top (PUSH and POP are the rather odd programmers' words for put-on-stack and take-from-stack respectively). This kind of stack (there are other, less common kinds) is called a 'LIFO stack' — LIFO stands for Last In,

```
0:    1   2   LOAD A;2       First No. to be multiplied
2:    11  2   LOAD X:2       Second number (hence 2 x 2')
4:    21      PUSH X         Save X temporarily
5:    20      PUSH A         Save A too
6:    13  98  STORE X;@98    We must still save X in memory
8:    11  98  LOAD X;98      ...for the indexed comparison
10:   7       SUB A;@X       Compare the old X with A
11:   8   17  JUMPNC;17      Jump if A exceeds the old X
13:   23      POP X          Put old A in X
14:   22      POP A          Put old X in A (swap)
15:   10  19  JUMP;19        Carry on
17:   22      POP A          Restore A (last in, first out)
18:   23      POP X          Restore X
                             Multiply starts here
19:   3   96  STORE A;@96    Save A, whatever it is now
21:   1   0   LOAD A;0       Fetch zero to...
23:   3   98  STORE A;@98    Clear the 'hundreds' total
25:   2   96  LOAD A;@96     Restore A
                             The 'adding loop' starts here
27:   16  1   SUB X;1        Count one less loop required
29:   9   34  JUMPNZ;34      Plug on unless we've reached 0
31:   3   99  STORE A;@99    Store the units
33:   0       HALT           That's all folks
34:   21      PUSH X         Save the count for later
35:   11  96  LOAD X;96      Point X at the multiplier
37:   19      ADD A;@X       Add it to the total yet again
38:   8   38  JUMPNC;38      Jump if it fitted in A
40:   20      PUSH A         It overflowed, save remainder
41:   2   98  LOAD A;@98     Fetch 'hundreds' so far
43:   5   1   ADD A;1        One more hundred
45:   3   98  STORE A;@98    Store the new 'hundreds' total
47:   22      POP A          Fetch remainder (units & tens)
48:   23      POP X          Get the count again
49:   10  27  JUMP;27        Do the next add if necessary
```

Listing 4. A better multiplier.

First Out, and it means simply that the last value PUSHed is always the first one POPped.

Since you can only access the 'top' item on the stack, the computer must have some way of knowing which item is top This is where the stack pointer comes in — it is an index register, rather like X, but it has the special property that its value falls by one whenever it is used to store something Some computers work the opposite way round, but the principle is the same. The TI-99/4A is the only well-known micro without a stack.

If you look at the 'multiply' program you see that we often have to 'save' and then 'restore' a register value. A stack is an ideal place to do this. Stack instructions can be short since they don't need an address (the stack pointer provides one). You can PUSH and POP numbers at will so long as you always retrieve them in the opposite order to that in which you saved them.

Listing 4 is a version of the 'multiply' program which uses PUSH and POP instructions to save and restore values. Notice that it is shorter and (hence) quicker than the original. It should also be easier to understand. If this explanation has seemed rather daft, type in the program and try it out. Look closely at the way the 'swap' works now.

```
0:    11   90    LOAD X;90        Point X to start
2:    17         LOAD A;@X        Fetch a number
3:    20         PUSH A           Save it on the stack
4:    15   1     ADD X;1          Point to next
6:    9    2     JUMPNZ;2         Round again unless finished
8:    11   99    LOAD X;90        Point back at the start
10:   22         POP A            Fetch last value pushed
11:   18         STORE A;@X       Dump it
12:   15   1     ADD X;1          Advance to the next
14:   9    10    JUMPNZ;10        Unless the end is nigh
16:   0          HALT             Job done
```

Listing 5. How to reverse values stored between 90 and 99.

STACKS OF PROBLEMS

Stacks can cause crashes! When you use Easycode the computer prints a message if you try to POP a value when S is 0 — the stack is empty Another message appears if you try to PUSH when S is 10 — the stack only has room for 10 items. On the popular small micros there is no check for over-running, or POPping values when the stack is empty — the computer just overwrites the location after the stack space, or fetches whatever is stored before the stack space. This is usually a disaster. Stacks are useful but they are also the cause of much confusion, so the more you experiment in the 'safe' environment of Easycode, the better

!ERAWTFOS ESREVER

Unlikely though it may sound, you will often need to reverse the order of values in machine-code programs. One example is the standard way of printing numbers, which involves successively dividing by 10 and printing the remainder This, unfortunately, gives the answer backwards:

```
236/10 = 23  remainder 6
 23/10 = 2   remainder 3
  2/10 = 0   remainder 2
```

but it is quite easy, using a stack, to reverse the order Listing 5 does the trick, reversing the values stored between locations 90 and 99.

DON'T PUSH

It may seem rather unfair to be allowed to PUSH the value of A or X, but not P (the usefulness of PUSH S is more debatable). In fact it would be quite useful to be able to PUSH the value of P. It would be a kind of bookmark — 'this is where I reached when I did the PUSH P'.

BASIC uses a kind of PUSH P arrangement to cope with the GOSUB statement, which has to remember 'where it came from' however many times it is used. GOSUB lets you use the same bit of program over and over again — wherever you call it from, a RETURN will always get you back. This is harder to program than you might imagine. You can't just use a 'where I came from' variable, because that won't handle GOSUBs within GOSUBs:

```
10 GOSUB 30
20 STOP
30 GOSUB 50
40 RETURN : REM (to 20)
50 RETURN : REM (to 40)
```

This program would never get back to line 20 if it used a single variable to store 'where I came from'. The GOSUB in line 30 would scrub the reference to line 20 stored by line 10 (we think!). The answer (as you probably guessed) is to use the dreaded LIFO stack, so that each RETURN matches the most recent GOSUB. And that's exactly what BASIC does, which is why a program like:

```
10 GOSUB 10
```

slowly eats up memory until you get an error message when the stack is full (usually something like 'Too many GOSUBs' or 'Out of Memory').

Now we've got a stack it is easy to add GOSUB and RETURN instructions, although assemblers tend to prefer words like CALL and JSR (Jump to SubRoutine) instead of GOSUB. Not being total masochists, we'll use CALL and RETURN. CALL saves the address of the following instruction (to avoid getting knotted up within a RETURN) on the stack RETURN fetches the last number off the stack and puts it into the 'P' register — in effect, JUMP,@S. To clarify this, try out this Easycode program which is identical in effect to the five-line BASIC program:

```
0: CALL;3
2: HALT
3: CALL;6
5: RETURN (to 2)
6: RETURN (to 5)
```

A PRACTICAL EXAMPLE

If you saved Listing 4 you can easily change it into a general-purpose multiplication subroutine. Just change the 0 at location 33 into a RETURN (code 25) You use the subroutine by loading A and X and then CALLing location 4. If you're sure that X will always be less than A, you could CALL,19 instead, skipping the swap instructions.

The last program of the month (Listing 6) assumes that you have entered Listing 4 and changed the HALT into a RETURN The program, which should be RUN from location 60, works out the square of the number in the A register (multiplying A by A). The result ends up in locations 98 and 99, as usual. This will be a vastly useful program next time you need to know how many sheets of paper are needed to cover the ceiling of a square room . . (always assuming you know the area of a sheet. a CT spread covers about 0.124 square metres!)

```
60:   1    1     LOAD A;3         Whatever size cell you prefer!
62:   4          LOAD A;X         Put the same number in X
64:   24   19    CALL;19          Do the multiplication
66:   0          HALT             Easy, wasn't it!
```

Listing 6. Storing A squared at 98, 99.

There is one crucial thing to remember about programming using the stack. You must clear away temporary results between a CALL and a RETURN — otherwise the computer could try to RETURN to the place A or X pointed to. You can't CALL a routine that goes PUSH A then RETURN — the value of A would 'get in the way' of the return address so that you'd end up 'returning' to whatever address happened to be in A. There are a few occasions in which this is likely to be useful, but not many!

The trick is to make sure that you always POP as much as you've PUSHed before returning from a subroutine. A common mistake is to 'save registers' before a CALL and then try to 'restore registers' inside the subroutine. This doesn't work either — the return address gets in the way.

EASYCODE PART 4

Siman N. Gaadwin

And so to the final act, where we introduce a few characters from the real world of microprocessors and compare their instruction sets to that of our BASIC simulation.

In this final instalment of the the 'Easycode' series we'll compare our invented machine-code with the real thing, and explain some of the features of common microprocessors. We've got examples in Easycode, Z80, 6502 and 6809 code, and we'll compare the performance of all of the popular processors.

THE DIFFICULT BIT

When this series began we deliberately skipped over an area which is normally introduced very early in machine-code tutorials — the idea of 'bits'. This isn't a very hard idea to grasp, but it is difficult to see why it is important when you've no background knowledge of machine code. By now you should know enough to be able to see how bits can be useful.

You've almost certainly been told, at one time or another, that computers store information in 'ones and noughts'. In other words, a voltage can be either present or absent at a point inside a computer or memory chip. If the voltage at a point is much over 2V we generally call the level 'Logic 1'. Less than a volt usually corresponds to 'Logic 0'. (Special purpose computers sometimes use different levels: officially, in micro systems, a '0' is represented by 0V and a '1' by 5V).

The important point is that a computer can't store other values. High voltages are '1's, lower voltages are '0's. The exact line between the two levels, or 'states', varies between components but it is generally between 1 and 2V. There is no 'Logic ½' level, and no 'Logic 3' or 'Logic −2'. Computer storage is composed of cells — or 'bits'. Each bit contains either the value '0' or '1'.

BASE 100

Easycode differs from real computers because each location — the smallest unit which can be handled — can contain any value in the range from 0 to 99. If you want to store a larger number in Easycode, you have to use more than one location. To store a value up to 9999 you use two locations, one for the 'units' and another for the 'hundreds'. In principle this could be extended indefinitely — to store the number of people in the UK you'd need four locations, for units, hundreds, tens of thousands, and millions. The limited capacity of an individual location doesn't matter much, so long as you can manipulate groups of them together.

BASE 2

A single bit is the most limiting capacity of all. You can use it to represent the answer to a 'yes/no' question but it isn't useful for much else on its own. Micro systems recognise this fact and most instructions deal with groups of bits rather than individual ones. An 'eight bit' computer is one which uses, most commonly, groups of eight bits, or 'bytes'.

Most popular micros use eight bit processors, although they often have a few facilities for handling information in larger or smaller amounts. There are jargon terms for each amount: 'words' (16 bits) — 'long words' (32 bits), 'pages' (often 256 bytes — 2084 bits), 'segments' (typically 65536 bytes) and lots of others, specific to individual processors. There are units smaller than a byte, too: a 'nibble' is two bits and a 'nybble' (note the 'i' and the 'y') is four bits.

The more bits there are in a unit, the more values can be represented. In a nibble you can store four different values: 00, 01, 10 and 11. This is a notation called 'binary' (meaning twofold) since each bit may be either a '0' or a '1'.

The number of values which can be stored in a certain number of bits is found by multiplying two (the number of possible states) by itself for each available bit. In two bits we can store four values, in four bits we can store 2 x 2 x 2 x 2 = 16 values, and so on. If you're not sure of this, check it out by writing all of the permutations of four bits. Once the rule is known we can say that 256 values will fit in a byte (eight bits), 65536 values will fit into a computer word (16 bits), and so on.

DECIMAL AND BINARY

It is helpful to be able to write numbers in binary, but sometimes they are rather unwieldy. It is much easier to say that the Spectrum's display memory is located between address 16384 and 23295, than to use the binary forms 100000000000000 and 101101011111111, even though the binary does represent the pattern of voltages used inside the computer! We need a consistent way to convert binary into decimal.

If we call the right hand digit the 'units', the next digit the 'twos', then the 'fours', 'eights', 'sixteens' and so on, we can convert from binary to decimal quite easily. The sequence of values ('powers of two' or 'weights'), from the right, goes 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 (a 'K'), 2048, 4096, 8192, 16384, 32768, 65536 and so on. You pick up this sequence quickly as you learn machine code.

TABLE 1

Weight:	128	64	32	16	8	4	2	1	
Binary:	1	0	1	0	1	0	1	0	
Decimal:	128	+	32	+	8	+	2	=	170

Binary to decimal conversion.

As an example, Table 1 converts the number 10101010 from binary to decimal. You should be able to confirm that 11111101 in binary is 253 in decimal, and 01010101 (binary) is 85 (decimal).

SHIFT WORK

There is a hidden advantage of binary arithmetic — it is better suited to electronic addition, multiplication, and so on. Even pocket calculators use binary internally, like computers, and then convert the values as required. You may have spotted the technique used when you tried out the examples above.

A microprocessor is much better at moving bits around than it is at complicated arithmetic operations like multiplication. There are electronic ways of multiplying or dividing in a single step, but they're certainly not trivial! What is easy is moving bits up and down in a register — so-called 'shifts' (if the end values are thrown away) or 'rotates' (if the end values re-appear at the other end).

Look back at the binary forms of 85 and 170. To convert 85 into 170 all you have to do is shift the binary one place to the left. Such a shift is, in effect, a 'multiply by 2' operation. Similarly a shift to the right corresponds to a 'divide by 2'. By convention the bit which 'falls off' after a shift or rotate (the remainder, here) is shunted into the carry flag.

Earlier in this series we performed a multiplication by adding a number to itself repeatedly. We warned that this was not the technique used in real computers — now we can see the real technique in action!

PROBLEMS MULTIPLYING

It is fairly easy to program a multiplication using a combination of shifts and additions (hardware addition is simpler than multiplication — all microprocessors support it). It becomes even easier once you realise that we're using binary, since the ones and noughts in the binary can be used to indicate when we should shift the result and when we should add the value to be multiplied.

As an example we'll multiply 23 by 11, using binary throughout. 11 decimal (the multiplier) is 1011 in binary, and 23 (the multiplicand, or 'other number') corresponds to 10111. The operations are 'dry run' in Table 2.

We end up with the binary value 11111101, which — amazingly enough — is 253 in decimal! This may seem like magic, but it does work reliably for any size of number, so long as you carry out one shift/rotate step for every bit in the multiplier

TABLE 2

STEP	ACTION	MULTIPLIER	RESULT
0	Set multiplier, clear result	1011	00000000
1	Shift result left (*)		00000000
	Rotate multiplier left. If a	0111	
	'1' fell off, add multiplicand		00010111
2	Shift result left		00101110
	Rotate multiplier left. If a	1110	
	'1' fell off, add multiplicand		
3	Shift result left		01011100
	Rotate multiplier left. If a	1101	
	'1' fell off, add multiplicand		01110011
4	Shift result left		11100110
	Rotate multiplier left. If a	1011	
	'1' fell off, add multiplicand		11111101

Binary multiplication.

If this still seems confusing, try thinking of it as a process of repeated additions with shifts intermingled. An add followed by four shifts corresponds to 16 adds, an add before two shifts corresponds to four adds, and so on. The later a bit appears in the multiplier, the less additions it represents. Each position represents half as many additions as the position to its left.

There is a 'useless' shift in step 1 — this shifts the value zero, making absolutely no difference! It has still been performed to emphasise that the procedure is absolutely identical at each step shift the result, shift the multiplier and add the multiplicand IF there was a carry from the multiplier). There should be as many steps as there is room for bits in the multiplier. Here we've used four bits, hence four steps are needed to line up the result correctly.

You may need more bits for the result than you did for the original number, just as in human arithmetic. In decimal, 99 times 90 (two-digit originals) gives a four-digit result. 8910, for those with boggling brains or flat calculator batteries! The number of digits (or bits) in the result is never more than the total number of digits in multiplier and multiplicand.

Table 2 presumes that you either know or can work out how to add binary numbers. The process is actually very simple, although computers are best at it so it is OK to skip the arithmetic as soon as you understand how it works! Binary addition is done from the right-hand digit leftwords, just as you were taught (with decimal) at school. A '1' can be 'carried' to the left as usual, so you have to take the previous 'carry' into account for every digit after the first. For each column of binary there are eight possibilities, shown in Table 3.

TABLE 3

0 + 0 +	No carry	= 0
0 + 0 +	carry	= 1
0 + 1 +	No carry	= 1
0 + 1 +	carry	= 0, carry 1
1 + 0 +	No carry	= 1
1 + 0 +	carry	= 0, carry 1
1 + 1 +	No carry	= 0, carry 1
1 + 1 +	carry	= 1, carry 1

In principle this rule can be used for addition inside the computer, although tricks are used so that the processor doesn't have to wait for all the rightmost bits to be added before it can work out the result (including carry) for the leftmost. These tricks are interesting, but rather beyond the scope of this article!

ILLOGICAL, CAPTAIN!

There is one more application of bits which it is useful to understand. This is the idea of 'logical operators', not spies from the planet Vulcan, but actions based on simple binary rules. Logical operators allow a programmer to pack different information into any bit or group of bits within a location. Shifts and rotates can be used to line the bits up before they are stored or recalled. Table 4 summarises the effects of the operators NOT, AND, OR and XOR.

The simplest logical operator is called 'NOT', 'inverting' or 'ones-complement'. The effect of the 'NOT' operation is to flip the values of each individual bit. Every one becomes a nought, and vice versa.

The other logical operators involve two numbers — the data, and a so-called 'mask' value which is used to determine the result. All of the operators are commutative, which means that you get the same result whichever value you say is the 'mask' — in normal arithmetic, addition is said to be commutative, since 7 + 2 = 2 + 7, whereas subtraction is not 7 − 2 is not the same as 2 − 7 (or so my bank manager insists!).

The AND operator produces a result which contains 'ones' at every position where the data AND the mask contained a '1'. This is useful when you want to check some of the bits in a number and ignore the others. To test whether a number is odd or even, for instance, you just AND it with 1. If the result is zero then the number is even, otherwise it contained a 'unit' and must be odd.

The OR operator produces a result which contains '1' at every position where the data OR the mask (or both) held a '1'. This is a useful way to set certain bits in a register. OR A;4 will set the bit third from the right (the 'fours' column) in register A, whether or not it was set before.

The last operator is the most devious. The 'exclusive OR' (XOR or EOR) operation sets a bit in the result whenever ONLY ONE of the corresponding bits in data and mask is set. XOR A;1 would make the contents of A odd if they were previously even, or vice versa. If A is an eight-bit register, XOR A,255 will produce the same result as NOT A — all of the bits in A will be flipped (since, for any bit, '1' XOR '1' is '0' and '0' XOR '1' is '1').

TABLE 4

Logical NOT:	
A	10011101
NOT A	01100010
Logical AND:	
A	10011101
B	11110000
A AND B	10010000
Logical OR:	
A	10011101
B	11110000
A OR B	11111101
Logical XOR:	
A	10011101
B	11110000
A XOR B	01101101

Logical operators.

The key thing to remember is that if you XOR a location with the same mask twice, you get the first number back. This is very useful for applications like graphics, when you want to store and then erase a pattern.

REAL MACHINE-CODES

This series ends with a look at the three most common forms of 'hobby' machine-code — Z80, 6502 and 6809 code. We'll also

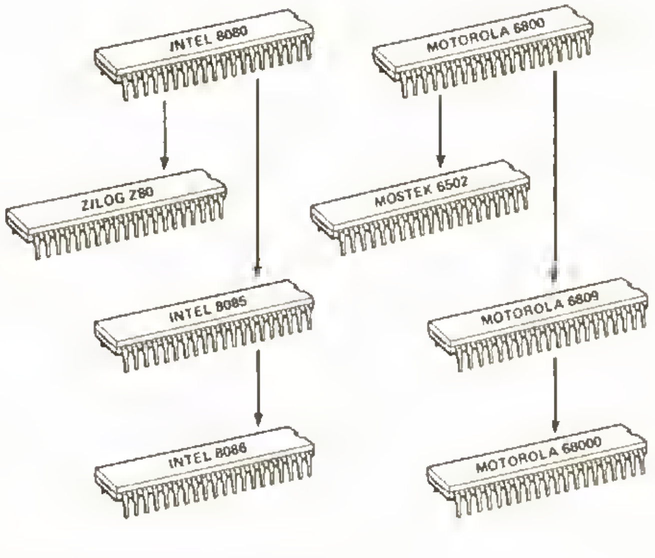


Fig. 1. The microprocessor family tree.

mention the 6800, 8080, 8085, 8086, 8088 and 68008 processors in passing!

There are, broadly speaking, two different types of eight bit microprocessor — Motorola-style and Intel-style. Motorola and Intel are both American chip-makers. Firms such as RCA, Texas and National Semiconductor have produced different designs, but they haven't caught on with hobby machine builders. Figure 1 shows a family tree of the most popular processors.

The first 'hit' micro was the Intel 8080, a development of an earlier calculator chip called the 8008. Later in 1975 Motorola produced the 6800 chip, a competing processor with a totally different internal design.

INTEL

Processors which follow the Intel design tend to have lots of registers and instructions, all of which have different, specialised applications.

The speed of a processor is determined by the timing clock signal it receives. Intel-style machines tend to take a number of pulses of the clock before they produce results (usually between four and 20 pulses, depending upon the instruction).

Common Intel-style processors are the 8080, 8085, 8086, 8088 and Z80. The 8085 and Z80 were both directly modelled on the 8080, to the extent that they share much of their instruction-set.

MOTOROLA

Motorola-style processors have relatively few registers and instructions. Instructions tend to be simple and consistent between registers, so that each register can be used in much the same way as the others (this is less true of the 6502).

Motorola established the idea of using 'instruction pre-fetch', which means that one part of the processor gets on with fetching information from memory while the other part decodes and performs the instructions. This 'parallel' principle is extended elsewhere in the processor, so that instructions take few clock pulses to be performed — generally between one and six pulses. This is one of the reasons why the Motorola-style BBC Micro (6502 at 2 million clock pulses a second, or 2 MHz) can out-perform the Lynx or Spectrum with their Intel-style Z80s running at twice the clock speed.

The Motorola 68008 processor used in Sinclair's QL computer has been criticised because it is a 'cut-down' version which addresses memory in bytes rather than 16-bit words. In fact the loss in performance is quite small — generally only about 30 per cent — because of the way the processor can access memory at the same time as internal operations are performed.

By way of contrast, many business machines use the Intel 8088 microprocessor, a cut-down model of the 16-bit 8086. The 8086 does not 'look ahead' in the same way, so the performance of the 8088 is degraded much more.

BATTLE OF THE BYTES

Listing 1 shows an Easycode program similar to one introduced earlier in this series. The program adds up a list of 10 numbers and stores the total in memory. Listings 2, 3 and 4 contain equivalent programs for the Z80, the 6502 and the 6809. The overall structure of the solution is the same in each case, although the detail has been altered. The programs are not claimed to be the best possible solution for each processor, but they do show the differences between the machines quite accurately.

Each real program starts with an ORG statement which tells the assembler where the code and data are to be stored. The value you pick depends upon the use of memory in your computer — it doesn't really matter where you pick so long as the memory is not already in use.

Words like DEFW, DW, and FDB are used to initialise a word location to a given value (zero, in the listings). Similarly DEFS, DS and RMB reserve a specified number of bytes for data (without explicitly storing a value). These words (called 'pseudo-ops' since they don't cause any processor operations) vary between processors and assemblers. The examples above were tested using EDAS on a Video Genie, AMAC on an Atari 800 and DASM on a Dragon 32.

Each assembler (other than Easycode) allows lines to be named or 'labelled'. This allows you to refer to data and code by name rather than address. On the Z80 and 6502 you define labels by following them with a colon — on the 6809 you prefix label names with an '@' sign.

```
 0:  11  81   LOAD X;81       Point to start of table+1
 2:   1   0   LOAD A;0        Fetch a zero to...
 4:   3  70   STORE A;@70     Clear the hundreds count
 6:   2  80   LOAD A;@80      Fetch first value
 8:  19       ADD A;@X        Add the next value
 9:   8  19   JUMPNC;19       Don't carry unless we must
11:  20       PUSH A          Save the units count
12:   2  70   LOAD A;@70      Fetch the hundreds
14:   5   1   ADD A;1         Count one more hundred
16:   3  70   STORE A;@70     Stor : the result
18:  22       POP A           Restore the units
19:  16  89   SUB X;89        Has X reached 89?
21:   9  26   JUMPNZ;26       Jump on if X was not 89
23:   3  71   STORE A;@71     We've finished, store units
25:  25       RETURN          Go back where you came from
26:  15  90   ADD X;90        Advance X to the next value
28:  10   8   JUMP;8          Go and add the next value
```

Listing 1. The Easycode adder (30 locations of code). It adds the values in 80-89, storing the result in 70 and 71.

THE Z80 PROGRAM

The Z80 processor is very popular with hobbyists, mainly because it has lots of 'special case' instructions. A very clever programmer can conseguently produce very clever programs! However, it does take a while to learn the idiosyncracies of the Z80.

In Z80 assembler a number or register-name in brackets represents a pointer to a location, so tht LD A, (100) would LoaD the contents of location 100 into register A (oddly, Z80 instructions move data from right to left!) and LD A,100 would put the value 100 into A. Some of the eight-bit registers can be used in pairs as 16-bit pointers: the H and L registers are used in this way in Listing 2. The DJNZ instruction is a typical compound Intel instruction. It combines a count-down (in the B register) with a conditional jump if the result is not zero.

```
         ORG     50000        ;Start assembly at address 50000
TOTAL:   DEFW    0            ;Room for the result (word value)
TABLE:   DEFS    10           ;Space for 10 data bytes

ADDUP:   LD      B,9          ;The number of 'adds' required
         LD      HL,TABLE     ;Registers H & L point at TABLE
         LD      C,0          ;C counts multiples of 256
         LD      A,(HL)       ;Fetch the first value into A
NEXT:    INC     HL           ;Point to the next value
         ADD     A,(HL)       ;A = A + what HL points at
         JR      NC,NOCAR     ;Rush on unless A overflowed
         INC     C            ;Add 1 to (increment) register C
NOCAR:   DJNZ    NEXT         ;Decrease B (B=B-1) Jump if not 0
         LD      L,A          ;Copy 'low' byte into register L
         LD      H,C          ;Copy 'high' byte to register H
         LD      (TOTAL),HL;Store result (H and L) in TOTAL
         RET                  ;Go West, young processor!
         END     ADDUP        ;This marks the start and end
```

Listing 2. The Z80 adder (22 locations of code). This adds the values in TABLE, storing the result in TOTAL.

THE 6502 PROGRAM

The 6502 is the most consistently-eight-bit processor, so Listing 3 has to store the 16-bit result in two eight-bit stages. Even the stack pointer is an eight-bit register, so a maximum of 256 bytes can be PUSHed. By way of compensation the 6502 has fast, useful

```
         ORG     1540         ;Start assembly at address 1540
TOTAL:   DW      0            ;Room for the result (word value)
TABLE:   DS      10           ;Space for 10 data values

ADDUP:   LDY     #1           ;Y selects each value in turn
         LDX     #0           ;X counts multiplies of 256
         LDA     TABLE        ;Put item at start of TABLE in A
NEXT:    ADC     TABLE,Y      ;Add value at TABLE+contents of Y
         BCC     NOCAR        ;Rush on unless A overflowed
         INX                  ;Add 1 to (increment) register X
NOCAR:   INY                  ;Point Y at the next value
         CPY     #10          ;Compare Y with 10
         BNE     NEXT         ;Branch (jump) to NEXT if Y<>10
         STX     TOTAL+1      ;Store X in high byte of TOTAL
         STA     TOTAL        ;Store A in low byte of TOTAL
         RTS                  ;We've finished (horray)
         END     ADDUP        ;This marks the start and end
```

Listing 3. The 6502 adder (25 locations of code).

instructions for 'indexed addressing', such as ADC TABLE,Y which takes the address TABLE, adds the eight-bit value in register Y, fetches a byte from the address totalled, and adds that byte to the value in the accumulator! In effect the instruction combines an adjustable array acess and an add in one step.

Motorola don't use brackets in the same way as the Z80 assembler. To load the VALUE 100 into register A you mark the number with a hash: LDA #100. To fetch the byte at address 100 use LDA 100.

Motorola-style computers call conditional jumps 'branches' to show that they use a trick called 'relative addressing'. The instruction doesn't contain the target address of the jump — instead it stores a positive or negative 'offset' between the following instruction and the one which might be fetched instead. Consequently BCC NOCAR is stored as 144 (the code for BCC) followed by '1' to indicate that one byte should be skipped (the INX) if the carry flag is clear. Relative branches make programs shorter and easier to move about in memory.

THE 6809 PROGRAM

It is a shame that the only common machines using the 6809 are the Dragon and Tandy Colour computers, which have weaknesses elsewhere in their design. The 6809 has lots of 16-bit instructions, an extra stack (so you don't need to get data and CALLs muddled up), even more ways of addressing data than the 6502, and a one-step built-in multiply operation. Despite its comprehensiveness, the instruction-set of the 6809 is the most consistent of any eight-bit processor, making it easy to learn and use efficiently. A seasoned CT contributor, Mike James, has written **The 6809 Companion** (Babani), which is an excellent cheap reference if you'd like to find out more for just £1.95.

Turning to our example program, four registers are used — the 16-bit index Y to point to the data, index X to store the total so far, A to count data items and B as a temporary store for each item.

```
        ORG     20480      ;Start assembly at address 20480
@TOTAL  FDB     0          ;Room for the result (word value)
@TABLE  RMB     10         ;Space for 10 data values

@ADDUP  LDA     #10        ;The A register counts the values
        LDX     #0         ;Clear result, which will be in X
        LDY     #@TABLE    ;Point register Y at the TABLE
@LOOP   LDB     0,Y+       ;Copy data at Y to B, add 1 to Y
        ABX                ;Add register B to total in X
        DECA               ;Count one less value to be added
        BNE     @LOOP      ;If A is not zero, go to LOOP
        STX     @TOTAL     ;Save the result at TOTAL
        RTS                ;Easy, eh?
        END     @ADDUP     ;This marks the start and end
```

Listing 4. The 6809 adder (19 locations of code).

"LDB 0,Y+" adds 0 to the contents of register Y (the zero is a dummy value in this case) and puts the contents of the total address into register B. The '+' tells the 6809 to add one to register Y (selecting the next byte in the table) while it is copying the byte into B! "ABX" is a rare but useful instruction which adds the 8-bit number in B to the 16-bit total in X.

CONCLUSION

With a little luck this series has explained the essence of machine code programming. If some of the details are a little unclear, don't worry — you learn programming by doing, not by reading. This series really set out just to encourage you to get started!

The next step is to buy, borrow or steal an assembler (make sure you get all of the instructions!) and buy a good book on your chosen processor (bad luck if you've got a TI 99/4A). The **Programming the 6502/6809/Z80** series by Rodnay Zaks (Sybex) will stand you in good stead, so long as you are careful not to buy first edition copies, which tend to be plastered with errors.

We look forward to seeing your concise machine-code masterpieces in CT soon!

PLEASE WAIT . . .
A Mystery Program

```
10 GOTO1880
20 REM
30 SCREEN1,1:CLS:CURSOR10,4:PRINT"Please wait":S=0
40 A=0:A=INT(RND(8)*5)+1:IFA=701THEN40
50 ONAGOTO60,310,380,460,550
60 SCREEN2,1:CLS:COLOR15,15,(0,0)-(255,191),15
70 LINE(10,10)-(245,181),8,B:LINE(10,10)-(122,90),,B
80 LINE(245,10)-(133,90),,B:LINE(245,181)-(133,101),,B
90 LINE(10,181)-(122,101),,B
100 LINE(115,11)-(115,65),4:LINE-(35,11)
110 LINE(140,11)-(140,65):LINE-(220,11)
120 LINE(140,180)-(140,126):LINE-(220,180)
130 LINE(115,180)-(115,126):LINE-(35,180)
140 LINE(11,81)-(82,81):LINE-(11,28)
150 LINE(244,81)-(173,81):LINE-(244,28)
160 LINE(244,110)-(173,110):LINE-(244,165)
170 LINE(11,110)-(82,110):LINE-(11,165)
180 LINE(11,11)-(106,81),8:LINE-(95,81):LINE-(11,19)
190 LINE(244,11)-(150,81):LINE-(140,81):LINE-(140,75):LINE-(230,11)
200 LINE(244,180)-(150,111):LINE-(163,111):LINE-(244,172)
210 LINE(11,180)-(105,110):LINE-(115,110):LINE-(115,116):LINE-(26,180)
220 PAINT(123,11),8:PAINT(11,29),4:PAINT(36,11)
230 PAINT(209,11):PAINT(244,29):PAINT(244,111):PAINT(141,127)
240 PAINT(113,128):PAINT(11,111)
250 PAINT(12,12),8:PAINT(240,11):PAINT(243,179):PAINT(18,179)
260 SCREEN2,2:FORA=0TO700:NEXTA
```

Easycode 32K Version for SEGA

```
10 REM AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
20 REM ** EASYCODE 32k version.
30 REM ** (C) 1984 Simon Goodwin
40 REM ** SEGA 1985 David Coursey
50 REM ** Tape routine data
60 DATA 2A,60,81,22,0C,98,2A,62,81
70 DATA 22,0E,98,2A,08,98,22,60,81
80 DATA 2A,0A,98,22,62,81,CD,69,7A
90 DATA 3E,0,32,A2,82,CD,9F,77
100 DATA 2A,0C,98,22,60,81,2A,0E,98
110 DATA 22,62,81,C9
120 DATA 2A,60,81,22,0C,98,2A,62,81
130 DATA 22,0E,98,2A,64,81,22,10,98
140 DATA 2A,66,81,22,12,98,2A,08,98
150 DATA 22,60,81,CD,EF,78,2A,0C,98
160 DATA 22,60,81,2A,62,81,22,66,81
170 DATA 2A,0E,98,22,62,81,2A,10,98
180 DATA 22,64,81,C9
190 REM ** Assembler text & codes
200 DATA STORE A;@X,18,LOAD A;@X,17
210 DATA STORE A;@,-3,STORE X;@,-13
220 DATA LOAD A;@,-2,LOAD A;X,4
230 DATA SUB A;@X,7,LOAD X;@,-12
240 DATA LOAD X;A,14,ADD A;@X,19,JUMPNZ;,-9
250 DATA JUMPNC;,-8,LOAD A;,-1,LOAD X;,-11
260 DATA PUSH A,20,PUSH X,21,RETURN,25
270 DATA ADD A;,-5,ADD X;,-15,SUB A;,-6
280 DATA SUB X;,-16,JUMP;,-10,POP A,22
290 DATA POP X,23,CALL;,-24,HALT,0
300 REM ** Screen display data
310 DATA 32,49,50,51,52,53,54,55,56,57,45,40,45
320 DATA 48,48,48,48,48,48,48,48,48,48,45,65,45
330 DATA 41,41,41,41,41,41,41,41,41,41,45,32,45
1000 TR=0:SD=0
1010 GOSUB 15500:REM Opening screen
1020 ERASE:REM Set variables to zero
1030 DIM R(1),M(99)
1040 MAX=25:REM Highest instruction code
1050 DIM D$(MAX),E(MAX),S(9)
1060 GOSUB 6500
1070 SCREEN 2,1:CLS
1080 GOTO 14000:REM Get menu
3490 REM ** Poll keys; Space=wait,<CR>=abort
3500 T$=INKEY$:REM keyboard scan
3510 IF T$=CHR$(13) THEN ABRT=1:BEEP
3520 IF T$<>CHR$(32) THEN RETURN
3530 BEEP
3540 ROW=15
3550 COLUMN=0
3560 GOSUB 6000:REM Position cursor on message line
3570 PRINT "Waiting at";P;
3580 PRINT " Press a key";
3590 FOR DE=1 TO 150:NEXT DE
3600 T$=INKEY$
3610 IF LEN(T$)=0 THEN 3600:REM No key yet,loop
3620 IF T$=CHR$(13)ORT$=CHR$(32) THEN 3640
3630 BEEP
3640 GOSUB 4000:REM Scrub the message
3650 GOTO 3510
3990 REM ** Clear line (leave cursor at start)
4000 GOSUB 6000:REM Position cursor
4010 PRINT CHR$(5)
4020 GOSUB 6000:REM Reset cursor
```

```
4030 RETURN
4490 REM ** Read number 0-99 to N (100=error)
4500 INPUT T$
4510 IF T$<"0"ORT$>=":" THEN 4570:REM Not digit
4520 N=VAL(T$)
4530 IF N<0  THEN 4570
4540 IF N>99 THEN 4570
4550 IF N<>INT(N) THEN 4570:REM F.P. Basic only
4560 RETURN:REM No error
4570 ROW=15
4580 COLUMN=0
4590 GOSUB 4000:REM Cursor to message line
4600 PRINT "* Number beyond range 0-99";
4610 N=100
4620 RETURN:REM Error found
4990 REM **Update display of registers and flags
5000 CARRY=0
5010 ZERO=0
5020 IF R(K)>=0 THEN 5050
5030 R(K)=R(K)+100
5040 GOTO 5070:REM Set carry
5050 IF R(K)<100  THEN 5080
5060 R(K)=R(K)-100
5070 CARRY=1
5080 IF R(K)=0 THEN ZERO=1
5090 ROW=12
5100 COLUMN=22
5110 GOSUB 6000:REM Set up for zero flag
5120 PRINT "N";
5130 GOSUB 6000:REM Position cursor
5140 IF ZERO=0 THEN PRINT "Y";
5150 COLUMN=28
5160 GOSUB 6000:REM Set up for carry  flag
5170 PRINT "N";
5180 GOSUB 6000:REM Position cursor
5190 IF CARRY=1 THEN PRINT "Y";
5200 COLUMN=1
5210 N=R(0)
5220 GOSUB 8000:REM Update accumulator display
5230 COLUMN=8
5240 N=R(1)
5250 GOSUB 8000:REM Update X register display
5260 GOTO 11500:REM Get next instruction
5490 REM ** Mark and update the current locn.
5500 ROW=INT(P/10)+1:REM F.PBasic only
5510 COLUMN=(P-10*ROW)*3+31
5520 GOSUB 6000:REM Put the cursor there
5530 PRINT "  ";:REM <2 SPC>
5540 ROW=12
5550 COLUMN=15
5560 N=P
5570 K=I:REM Save instruction code
5580 GOSUB 8000:REM Update program    counter
5590 N=M(P)
5600 I=P
5610 GOSUB 16500:REM Redraw curent    location
5620 GOSUB 3500:REM Poll the keyboard
5630 I=K:REM Restore instruction code
5640 RETURN
5990 REM ** Position cursor at column & row
6000 CURSOR COLUMN,ROW:PRINT "";
6010 RETURN
6490 REM Set up data for Assm. & Dism.
6500 RESTORE 190
6510 FOR I=0 TO 25
6520 READ D$(I),E(I):REM Text & instruction No.
6530 NEXT I
6540 RETURN
6990 REM ** Push N onto stack
7000 COLUMN=33
7010 ROW=10-STACK
```

```
7020 IF ROW=0 THEN 7110
7030 S(STACK)=N
7040 GOSUB 8000
7050 STACK=STACK+1
7060 COLUMN=34
7070 ROW=12
7080 N=STACK
7090 GOSUB 8000:REM Update S display
7100 RETURN
7110 COLUMN=0
7120 ROW=15
7130 GOSUB 6000:REMPrepare for message
7140 PRINT "* Stack full";
7150 GOTO 11600:REM Leaves 1 GOSUB stacked
7490 REM ** POP N from top of stack
7500 STACK=STACK-1
7510 IF STACK<0 THEN 7590:REM Whoops, error
7520 GOSUB 7060
7530 COLUMN=33
7540 ROW=10-STACK
7550 GOSUB 6000:REM Prepare to clear  old entry
7560 PRINT "  ";:REM <2 SPC>
7570 N=S(STACK)
7580 RETURN
7590 PRINT "* Nothing left on stack";
7600 GOTO 11600
7990 REM ** Print N at current coordinates
8000 GOSUB 6000
8010 IF N>9 THEN T$=STR$(N):L=LEN(T$):T$=RIGHT$(T$,L-1)
8020 IF N<10 THEN T$=STR$(N)+" ":REM Force two character width
8030 PRINT MID$(T$,1,2);:REM In range 0-99
8040 RETURN
8090 REM ** Print K at current coordinates
8100 GOSUB 6000
8110 IF K>9 THEN T$=STR$(K):L=LEN(T$):T$=RIGHT$(T$,L-1)
8120 IF K<10 THEN T$=STR$(K)+" ":REM Force two character width
8130 PRINT MID$(T$,1,2);:REM In range 0-99
8140 RETURN
8990 REM ** Draw the display
9000 I=0
9010 CLS
9020 GOSUB 9500:REM 2LH,1RH column
9030 FOR ROW=1 TO 10
9040 COLUMN=0
9050 GOSUB 6000:REM Position cursor
9060 PRINT ":"
9070 FOR J=1 TO 10
9080 COLUMN=J*3-2
9090 N=M(I)
9100 GOSUB 8000:REM Print one memory  element
9110 I=I+1
9120 NEXT J:PRINT"(":NEXT ROW
9130 ROW=11
9140 COLUMN=0
9150 GOSUB 6000:REM Position cursor
9160 FOR I=0 TO 36
9170 PRINT "-";
9180 NEXT I
9190 ROW=12
9200 GOSUB 6000:REM Position cursor
9210 PRINT "=00).(X=00).(P=00).(Z=N).(C=N).(5=00)"
9220 ROW=13
9230 GOSUB 6000:REM Position cursor
9240 FOR I=0 TO 36
9250 PRINT "-";
9260 NEXT I
9270 ROW=15
9280 GOSUB 6000:REM Cursor on message line
9290 COLUMN=0
9300 ROW=21
9310 GOSUB 6000:REM Cursor on message line
```

```
9320 PRINT "? for HELP"
9330 RETURN
9490 REM ** 2*LH,1*RH screen columns
9500 X=0
9510 RESTORE 310
9520 FOR Y=1 TO 13:READ B
9530 VPOKE Y*40+X+&H3C00,B
9540 NEXT
9550 X=X+1:IF X=2 THEN X=39
9560 IF X=40 THEN 9580:REM Job done
9570 GOTO 9520:REM Carry on
9580 RETURN:REM 9030
9990 REM ** Get the user's next command
10000 ROW=14
10010 COLUMN=0
10020 GOSUB 4000:REM Clear prompt line
10030 PRINT "Command";
10040 INPUT"> "; T$
10050 REM ** Force capitals
10060 C2$=""
10070 FOR C=1 TO LEN(T$):C1$=MID$(T$,C,1)
10080 IF C1$>="a" THEN C1$=CHR$(ASC(C1$)-32)
10090 C2$=C2$+C1$:NEXT
10100 T$=RIGHT$(C2$,LEN(T$))
10110 GOSUB 10130
10120 GOTO 10180
10130 ROW=15
10140 GOSUB 4000:REM Clear message line
10150 ROW=16
10160 GOSUB 4000:REM Clear extra line
10170 RETURN
10180 IF T$="RUN"  THEN 11000
10190 IF T$="SAVE" THEN 12000
10200 IF T$="LOAD" THEN 13000
10210 IF T$="?" THEN 14330
10220 IF T$="QUIT" THEN 10380
10230 IF T$="CLEAR"  THEN 10400
10240 IF T$="STORE" THEN 16000
10245 IF T$="HCOPY" THEN 22000
10250 IF T$="DIS" THEN 17000
10260 IF T$="ASM" THEN 18000
10270 PRINT "* ";T$;" is not a valid command";
10280 GOTO 10000
10290 REM ** Affirm QUIT/CLEAR command
10300 COLUMN=0
10310 ROW=15
10320 GOSUB 4000
10330 PRINT "Discard current workspace (Y/N)";
10340 INPUT T$
10350 IF T$="Y" THEN RETURN
10360 IF T$="N" THEN 12260
10370 IF T$<>"Y"OR T$<>"N" THEN 10350
10380 GOSUB 10300:REM Print prompt
10390 GOTO 15000:PRINT MENU
10400 GOSUB 10300:REM Print prompt
10410 CURSOR 0,14:PRINT "Working:"
10420 COLUMN=0
10430 ROW=15
10440 GOSUB 4000
10450 I=0
10460 FOR ROW=1TO10
10470 FOR J=1TO10
10480 IF I=99 THEN 10510:REM Last locn
10490 IF M(I)<>0 THEN 10540
10500 IF M(I)=0THEN 10570:REM Carry on
10510 COLUMN=0
10520 GOSUB 9190:REM Clear registers
10530 GOTO 1020:REM Clear variables
10540 COLUMN=J*3-2
10550 N=0
10560 GOSUB 8000:REM Draw at mem. locn
```

```
10570 I=I+1
10580 NEXT J:NEXT ROW
10590 REM ** Program has been stopped
10600 ROW=15
10610 COLUMN=0
10620 GOSUB 6000:REM Prepare for message
10630 PRINT "* Program stopped";
10640 GOTO 10000:REM Get a command
10990 REM ** "RUN" command pre-processor
11000 COLUMN=0
11010 ROW=14
11020 GOSUB 4000:REM Clear the prompt line
11030 PRINT "Start address";
11040 GOSUB 4500:REM Get the start of the program
11050 IF N>99 THEN 10000:REM Error
11060 P=N:REM Set program counter
11070 ABRT=0:REM Clear the abort flag
11080 COLUMN=33:REM Clear stack
11090 FOR ROW=1 TO 10
11100 GOSUB 6000:REM Position cursor
11110 PRINT "  ";:REM <2 SPC>
11120 NEXT ROW
11130 STACK=0
11140 N=STACK
11150 COLUMN=34
11160 ROW=12
11170 GOSUB 8000:REM Rewrite stack pointer
11180 K=0
11190 GOTO 5000:REM Write A,X etc
11490 REM ** "RUN" main loop for each instruction
11500 I=M(P):REM Get next instruction
11510 GOSUB 5500:REM Update display,  check keys
11520 IF ABRT=1 THEN 10600:REM Quit if requested
11530 COLUMN=0
11540 ROW=15
11550 GOSUB 6000:REM Put cursor on    message line
11560 IF I<1 THEN 11580:REM Halt code
11570 IF I<=MAX THEN 11620:REM Other instruction
11580 IF I=0 THEN PRINT "HALT";
11590 IF I<>0 THEN PRINT "* Unknown instruction";
11600 PRINT " at";P;
11610 GOTO 10000:REM Get next command
11620 IF P<>99 THEN 11650:REM Not end of memory
11630 PRINT "* No end on program";
11640 GOTO 10000:REM Get next command
11650 P=P+1
11660 J=M(P):REM Get operand
11670 P=P+1:REM Point to next instruction
11680 K=0:REM Assume a register A instruction
11690 IF I=21 OR I=23 THEN K=1
11700 IF I>10 THEN IF I<17 THEN K=1:REM Wrong ! Register X
11710 ON I GOTO 20000,20100,20200,20300,20400,20500,20600,20700,20800,20900,200
0,20100,20200,20300,20400,20500,21000,21100,21200:REM Execute instructions
11720 ON I-19 GOTO 21300,21300,21400,21400,21500,21600
11990 REM ** "SAVE" current program
12000 GOSUB 12500:REM Get Filename
12010 POKE&H9808,PEEK(&H8162):POKE&H9809,PEEK(&H8163)
12020 POKE&H980A,PEEK(&H8166):POKE&H980B,PEEK(&H8167)
12030 COLUMN=0
12040 ROW=15
12050 GOSUB 4000:REMClear message line
12060 PRINT "CONTINUE SAVE (Y/N)";
12070 INPUT T$
12080 IF T$="Y" THEN 12110
12090 IF T$="N" THEN 12260
12100 IF T$<>"Y"OR T$<>"N" THEN 12080
12110 COLUMN=0
12120 ROW=15
12130 GOSUB 4000:REMClear message line
12140 PRINT "Press SAVE & LOAD then <CR>";
12150 INPUT "*";T$
```

```
12160 COLUMN=0
12170 ROW=15
12180 GOSUB 4000:REMClear message line
12190 PRINT "Saving ";FL$
12200 CALL&H9814
12210 CURSOR0,17:PRINT "                    "
12220 CURSOR0,18:PRINT "                    "
12230 CURSOR0,19:PRINT "                    "
12240 CURSOR0,20:PRINT "                    "
12250 CURSOR0,15:PRINT "Saving end     "
12260 GOSUB 10130:REM Clear extra line
12270 GOTO 10000:REM Get next command
12490 REM ** Get filename
12500 ROW=14
12510 COLUMN=0
12520 GOSUB 4000
12530 PRINT "Enter Filename * ";
12540 INPUT"EASY.";T$
12550 FL$="EASY."+T$
12560 FOR BZ=LEN(FL$) TO 16:REM Pad   filename with blanks
12570 FL$=FL$+CHR$(32):NEXT
12580 RETURN
12990 REM ** 'LOAD' memory from tape
13000 COLUMN=0
13010 ROW=15
13020 GOSUB 4000
13030 PRINT "Destroy existing data (Y/N)";
13040 INPUT T$
13050 IF T$="Y" THEN 13080
13060 IF T$="N" THEN 12080
13070 IF T$<>"Y"OR T$<>"N" THEN 13050
13080 COLUMN=0
13090 ROW=15
13100 GOSUB 4000
13110 COLUMN=0
13120 ROW=14
13130 GOSUB 6000
13140 PRINT "Working:"
13150 POKE&H9808,PEEK(&H8162):POKE&H9809,PEEK(&H8163)
13160 POKE&H82A2,0
13170 CALL&H9844
13180 CURSOR 0,17:PRINT "                               "
13190 CURSOR 0,16:PRINT " Loading end  "
13200 I=0
13210 FOR ROW=1TO10
13220 FOR J=1TO10
13230 IF I=99 THEN 13260
13240 IF M(I)<>0 THEN 13290
13250 IF M(I)=0 THEN 13320
13260 COLUMN=0
13270 GOSUB 9190
13280 GOTO 10000
13290 COLUMN=J*3-2
13300 N=M(I)
13310 GOSUB 8000
13320 I=I+1
13330 NEXT J:NEXT ROW:P=0
13490 REM ** Tape routine Mcode loader
13500 RESTORE 60
13510 FOR X=&H9814 TO &H987D
13520 READ A$:POKEX,VAL("&H"+A$):NEXT
13530 RETURN:REM to 15620
13990 REM ** 'HELP' command received
14000 SCREEN 2,1:REM Change the screen
14010 COLOR 1,3,(0,0)-(255,191),3:M1=1
14020 PRINT"     Valid EASYCODE commands are:
14030 PRINT
14040 PRINT"     STORE to enter data on program"
14050 PRINT"     CLEAR to reset MON+ memory"
14060 PRINT"     RUN   to execute a MON+ program"
14070 PRINT"     SAVE  to store one on tape"
```

```
14080 PRINT"     LOAD  to read one from tape"
14090 PRINT"      ?    to view this message"
14100 PRINT"     QUIT  to return to Basic"
14110 PRINT"     HCOPY to print working screen":PRINT
14120 PRINT"     DIS   to disassemble memory"
14130 PRINT"     ASM   to assemble into memory"
14150 CURSOR 20,160
14160 PRINT"       Loading Working Screen"
14170 IF SD=0 THEN 14240
14180 CURSOR 10,160:PRINT CHR$(5)
14190 CURSOR 10,160:PRINT"       Press space bar to continue";
14200 SCREEN 2,2
14210 IF INKEY$<>CHR$(32)THEN 14210
14220 CURSOR 0,15:PRINT "             "
14230 GOTO 14300
14240 SCREEN 1,1
14250 GOSUB 15650:REM Display menu?
14260 SD=SD+1:SCREEN 1,2
14270 GOSUB 9000:REM Redraw display
14280 SCREEN 2,2
14290 GOTO 14180
14300 SCREEN 1,1
14310 CURSOR 0,15:PRINT "             "
14320 GOTO 10000:REM Get next command
14330 IF M1=1 THEN 14200
14340 CURSOR 0,15:PRINT "Loading Menu"
14350 SCREEN 2,1:CLS:M1=1:M2=0
14360 GOTO 14020
14990 REM ** 'QUIT' routine (nice and simple!)
15000 CLS
15010 END:REM That's all folks
15490 REM ** Opening screen
15500 SCREEN 1,1:CLS
15510 CURSOR 10,0:PRINT "E A S Y C O D E"
15520 CURSOR 13,4:PRINT "Simulated"
15530 CURSOR 11,5:PRINT "Machine  code"
15540 CURSOR 14,6:PRINT "Monitor"
15550 CURSOR 14,9:PRINT "for 32k"
15560 CURSOR 12,11:PRINT "SEGA SC3000"
15570 CURSOR 3,15:PRINT "Copyright (C) 1984 Simon Goodwin."
15580 CURSOR 3,16:PRINT "SEGA  version 1985 David Coursey."
15590 IF TR>0 THEN 15630
15600 CURSOR 9,19:PRINT "Loading Mcode data"
15610 GOSUB 13500
15620 TR=TR+1
15630 CURSOR 5,19:PRINT "       Loading Menu               "
15640 RETURN:REM to 1020
15650 CURSOR 5,19:PRINT "Press space bar to continue"
15660 IF INKEY$<>CHR$(32) THEN 15660
15670 RETURN:REM 14260
15990 REM ** 'STORE' data or program
16000 COLUMN=0
16010 ROW=15
16020 GOSUB 4000:REM Clear messages   (for later)
16030 ROW=14
16040 GOSUB 4000:REM Clear prompt line
16050 PRINT "Enter address (100 to stop)";
16060 GOSUB 4500:REM Get number
16070 IF N>99 THEN 10000:REM Error
16080 K=N
16090 ROW=15
16100 COLUMN=0
16110 GOSUB 4000:REMSet up next prompt
16120 PRINT "Enter data (100 to stop)";
16130 ROW=14
16140 GOSUB 4000:REM Set up varying   prompt
16150 PRINT"Address";K;"=";
16160 GOSUB 4500:REM Get number
16170 IF N>99 THEN 16000:REM Error
16180 I=K
16190 GOSUB 16500:REM Store in memory & display
```

```
16200 K=K+1:REM Select next location
16210 IF K<100 THEN 16090:REM Get more
16220 ROW=15
16230 COLUMN=0
16240 GOSUB 4000:REM Clear old message
16250 PRINT "* End of memory reached";
16260 GOTO 10000:REM Get new command
16490 REM ** Put value N in M() and on screen
16500 M(I)=N
16510 ROW=INT(I/10)+1:REM F.P Basic only
16520 COLUMN=(I-10*ROW)*3+31
16530 GOSUB 8000:REM Print the number
16540 RETURN
16990 REM ** Dissasembler - main loop
17000 ROW=14
17010 COLUMN=0
17020 GOSUB 4000:REM Clear prompt line
17030 PRINT "Disassemble from";
17040 GOSUB 4500:REM Get number
17050 IF N>99 THEN 10000:REM Whoops
17060 SCREEN 2,1:CLS:SCREEN 2,2:M1=0:M2=0
17070 PRINT "   Addr...Value....Disassembly";
17080 FOR P=16 TO 112 STEP 8
17090 GOSUB 17500:REM Disassemble 1 line
17100 NEXT P
17110 PRINT:PRINT:PRINT
17120 PRINT "   Continue disassembly (Y/N)";
17130 IF INKEY$<>"N" THEN 17150
17140 GOTO 17180
17150 IF INKEY$<>"Y" THEN 17130
17160 N=N-2:REM Ensure overlap
17170 CLS:GOTO 17070
17180 IF SD=0 THEN CLS:GOTO 17200
17190 CLS:GOTO 17220
17200 SD=SO+1:SCREEN 1,1
17210 GOSUB 9000:REM Redraw screen
17220 SCREEN 1,1
17230 GOTO 10000:REM Back to command
17490 REM ** Oisassemble the code at N
17500 IF N>99 THEN RETURN:REM End of  memory
17510 K=M(N):REM Get 1 char
17520 COLUMN=28
17530 ROW=P
17540 GOSUB 8000:REM Address field
17550 COLUMN=74
17560 GOSUB 8100:REM Value field
17570 J=100
17580 FOR I=0 TO 25
17590 IF K<>ABS(E(I)) THEN 17620
17600 J=I:REM Get instruction text No.
17610 I=100:REM Flag end of loop
17620 NEXT I
17630 IF J>25 THEN 17800
17640 IF E(J)<0 THEN 17700:REM 2 Char.instruction
17650 COLUMN=128
17660 GOSUB 6000:REM Instruction field
17670 PRINT D$(J);
17680 N=N+1:REM Select next address
17690 RETURN
17700 N=N+1
17710 IF N>99 THEN 17800
17720 K=M(N)
17730 COLUMN=88
17740 GOSUB 8040
17750 COLUMN=128
17760 GOSUB 6000:REM Instruction field
17770 PRINT O$(J);
17780 PRINT K;
17790 GOTO 17680
17800 COLUMN=128
17810 GOSUB 6000:REM Instruction field
```

```
17820 PRINT "Data code:";K;
17830 GOTO 17680:REM Exit
17990 REM ** ASSEMBLE - main loop
18000 ROW=14
18010 COLUMN=0
18020 GOSUB 4000:REM Cursor for prompt
18030 PRINT "Assemble to";
18040 GOSUB 4500:REM Get address
18050 IF N>99 THEN 10000:REM Error
18060 K=N:REM Save start address
18070 ROW=15
18080 GOSUB 4000:REM Cursor to messageline
18090 PRINT "Assembling. Type 100 to stop";
18100 ROW=14
18110 GOSUB 4000
18120 PRINT K;"=";
18130 INPUT T$
18140 IF T$="?" THEN 19270
18150 IF T$="100" THEN 18190
18160 GOSUB 18500:REM Assemble 1 line
18170 COLUMN=0:REM Just in case
18180 IF K<100  THEN 18070:REM Re-prompt
18190 ROW=14
18200 GOSUB 4000
18210 ROW=15
18220 GOSUB 4000
18230 IF K>99 THEN PRINT "* End of memory reached";
18240 GOTO 10000:REM Get new command
18490 REM ** Assemble one line into M(K) from T$
18500 J=100
18510 FOR I=0 TO 25
18520 IF D$(I)<>LEFT$(T$,LEN(D$(I))) THEN 18570
18530 J=E(I):REM Get instruction code
18540 I=LEN(D$(I)):REM Get length of  instruction
18550 T1$=RIGHT$(T$,LEN(T$)-I):REM  Get remainder
18560 I=100:REM Flag end of loop
18570 NEXT I
18580 IF J<0 THEN 18760:REM 2 char.   instruction
18590 IF J<26 THEN 18680:REM 1 char.  instruction
18600 T$="* Unknown:"+T$
18610 ROW=15
18620 GOSUB 4000:REM Clear old message
18630 PRINT T$;
18640 FOR I=0 TO 1500
18650 NEXT I
18660 GOSUB 4000:REM Clear message
18670 RETURN:REM Error exit
18680 IF T1$="" THEN 18710:REM No trailing junk
18690 T$="* Too long:"+T$
18700 GOTO 18610:REM Print message and return
18710 N=J
18720 I=K
18730 GOSUB 16500:REM Update display
18740 K=K+1:REM 1 more location used
18750 RETURN:REM Success return
18760 IF K<99 THEN 18790
18770 T$="* Only 1 memory space left"
18780 GOTO 18610:REM Print message
18790 IF T1$<"0" OR T1$>"99" THEN 18860
18800 I=VAL(T1$):REM Check range 0-99
18810 IF I>99 THEN 18860
18820 N=ABS(J):REMGet instruction code
18830 GOSUB 18720:REM Store N
18840 N=VAL(T1$)
18850 GOTO 18720:REM Store parameter &return
18860 T$="* Incorrect number:"+T1$
18870 GOTO 18610:REM Report error
18990 REM ** HELP for Assembler user
19000 SCREEN 2,1:CLS:M1=0:M2=1
19010 PRINT "        Valid instructions:";
19020 FOR R=0 TO 95 STEP 8
```

```
19030 ROW=R+16
19040 COLUMN=40
19050 GOSUB 6000
19060 PRINT D$(R/8);
19070 IF E(R/8)<0 THEN PRINT "nn";
19080 COLUMN=144
19090 GOSUB 6000
19100 PRINT D$(R/8+13);
19110 IF E(R/8+13)<0 THEN PRINT "nn";
19120 NEXT R
19130 PRINT:PRINT:PRINT "     nn is a number from 0 to 99"
19140 PRINT:PRINT:PRINT:PRINT "     Press space bar to continue";
19150 SCREEN 2,2
19160 IF INKEY$<>CHR$(32) THEN 19160
19170 IF SD=0 THEN GOTO 19190
19180 GOTO 19210
19190 SD=SD+1:SCREEN 1,1:CLS
19200 GOSUB 9000:REM Redraw screen
19210 SCREEN 1,1
19220 ROW=16
19230 COLUMN=0
19240 GOSUB 4000
19250 COLUMN=0
19260 GOTO 18070
19270 IF M2=1 THEN 19150
19280 CURSOR 0,15:PRINT "Loading Menu
19290 GOTO 19000
19300 REM ** LOAD Register;number
19310 R(K)=J
19320 GOTO 5000:REM Set flags & update display
19990 REM ** LOAD Register;number
20000 R(K)=J
20010 GOTO 5000:REM Set flags & update display
20090 REM ** LOAD Register;memory
20100 R(K)=M(J)
20110 GOTO 5000
20190 REM ** STORE Register;memory
20200 I=J
20210 N=R(K)
20220 GOSUB 16500:REM Display alteration
20230 GOTO 11500:REM No flags - just  get next
20290 REM ** LOAD Register;Register'
20300 R(K)=R(1-K)
20310 P=P-1:REM Only a 1 char. instruction
20320 GOTO 5000
20390 REM ** ADD Register;number
20400 R(K)=R(K)+J
20410 GOTO 5000
20490 REM ** SUB Register;number
20500 R(K)=R(K)-J
20510 GOTO 5000
20590 REM ** SUB A;@X
20600 R(0)=R(0)-M(R(1))
20610 P=P-1:REM Only a 1 char. instruction
20620 GOTO 5000
20690 REM ** JUMPNC;address
20700 IF CARRY=0 THEN P=J
20710 GOTO 11500
20790 REM ** JUMPNZ;address
20800 IF ZERO=0 THEN P=J
20810 GOTO 11500
20890 REM ** JUMP;address
20900 P=J
20910 GOTO 11500
20990 REM ** LOAD A;@X
21000 R(0)=M(R(1))
21010 P=P-1:REM 1 char. instruction
21020 GOTO 5000
21090 REM ** STORE A;@X
21100 N=R(0)
21110 I=R(1)
```

```
21120 P=P-1:REM 1 char. instruction
21130 GOSUB 16500:REM Store & display
21140 GOTO 11500
21190 REM ** ADD A;@X
21200 R(0)=R(0)+M(R(1))
21210 P=P-1
21220 GOTO 5000
21290 REM ** PUSH Register
21300 N=R(K)
21310 GOSUB 7000:REM Put N on stack
21320 P=P-1
21330 GOTO 11500
21390 REM ** POP Register
21400 GOSUB 7500:REM Get N from stack
21410 R(K)=N
21420 P=P-1
21430 GOTO 5000
21490 REM ** CALL;Address
21500 N=P:REM Save current program counter
21510 GOSUB 7000:REM Push address
21520 P=J:REM Start processing there
21530 GOTO 11500
21590 REM ** RETURN
21600 GOSUB 7000:REM Get return address
21610 P=N:REM Start processing there
21620 GOTO 11500
21990 REM ** HCOPY
22000 IN=1
22010 FOR VP=&H3C00 TO &H3E7F
22020 VD=VPEEK(VP)
22030 LPRINT CHR$(18):LPRINT "S1"
22040 LPRINT "P";CHR$(VD);
22050 IF IN=40 THEN GOSUB 22100
22060 IN=IN+1
22070 NEXT
22080 LPRINT"A":GOTO 10000
22100 LPRINT CHR$(18):LPRINT "A":LPRINT CHR$(18):IN=0
22110 RETURN
```

Easycode 16K Version for SEGA

```
10 REM AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
20 REM ** EASYCODE 16k version.
30 REM ** (C) 1984 Simon Goodwin
40 REM ** SEGA 1985 David Coursey
50 REM ** Tape routine data
60 DATA 2A,60,81,22,0C,98,2A,62,81
70 DATA 22,0E,98,2A,08,98,22,60,81
80 DATA 2A,0A,98,22,62,81,CD,69,7A
90 DATA 3E,0,32,A2,82,CD,9F,77
100 DATA 2A,0C,98,22,60,81,2A,0E,98
110 DATA 22,62,81,C9
120 DATA 2A,60,81,22,0C,98,2A,62,81
130 DATA 22,0E,98,2A,64,81,22,10,98
140 DATA 2A,66,81,22,12,98,2A,08,98
150 DATA 22,60,81,CD,EF,78,2A,0C,98
160 DATA 22,60,81,2A,62,81,22,66,81
170 DATA 2A,0E,98,22,62,81,2A,10,98
180 DATA 22,64,81,C9
300 REM ** Screen display data
310 DATA 32,49,50,51,52,53,54,55,56,57,45,40,45
320 DATA 48,48,48,48,48,48,48,48,48,48,45,65,45
1000 SD=0
1010 GOSUB 15500:REM Opening screen
1020 ERASE:REM Set variables to zero
1030 DIM R(1),M(99)
1040 MAX=19:REM Highest instruction code
1070 SCREEN 2,1:CLS
```

```
1080 GOTO 14000:REM Get menu
3490 REM ** Poll keys; Space=wait,<CR>=abort
3500 T$=INKEY$:REM keyboard scan
3510 IF T$=CHR$(13) THEN ABRT=1:BEEP
3520 IF T$<>CHR$(32) THEN RETURN
3530 BEEP
3540 ROW=15
3550 COLUMN=0
3560 GOSUB 6000:REM Position cursor on message line
3570 PRINT "Waiting at";P;
3580 PRINT " Press a key";
3590 FOR DE=1 TO 150:NEXT DE
3600 T$=INKEY$
3610 IF LEN(T$)=0 THEN 3600:REM No key yet,loop
3620 IF T$=CHR$(13)ORT$=CHR$(32) THEN 3640
3630 BEEP
3640 GOSUB 4000:REM Scrub the message
3650 GOTO 3510
3990 REM ** Clear line (leave cursor at start)
4000 GOSUB 6000:REM Position cursor
4010 PRINT CHR$(5)
4020 GOSUB 6000:REM Reset cursor
4030 RETURN
4490 REM ** Read number 0-99 to N (100=error)
4500 INPUT T$
4510 IF T$<"0"ORT$>=":" THEN 4570:REM Not digit
4520 N=VAL(T$)
4530 IF N<0  THEN 4570
4540 IF N>99 THEN 4570
4550 IF N<>INT(N) THEN 4570:REM F.P. Basic only
4560 RETURN:REM No error
4570 ROW=15
4580 COLUMN=0
4590 GOSUB 4000:REM Cursor to message line
4600 PRINT "* Number beyond range 0-99";
4610 N=100
4620 RETURN:REM Error found
4990 REM **Update display of registers and flags
5000 CARRY=0
5010 ZERO=0
5020 IF R(K)>=0 THEN 5050
5030 R(K)=R(K)+100
5040 GOTO 5070:REM Set carry
5050 IF R(K)<100  THEN 5080
5060 R(K)=R(K)-100
5070 CARRY=1
5080 IF R(K)=0 THEN ZERO=1
5090 ROW=12
5100 COLUMN=22
5110 GOSUB 6000:REM Set up for zero flag
5120 PRINT "N";
5130 GOSUB 6000:REM Position cursor
5140 IF ZERO=0 THEN PRINT "Y";
5150 COLUMN=28
5160 GOSUB 6000:REM Set up for carry  flag
5170 PRINT "N";
5180 GOSUB 6000:REM Position cursor
5190 IF CARRY=1 THEN PRINT "Y";
5200 COLUMN=1
5210 N=R(0)
5220 GOSUB 8000:REM Update accumulator display
5230 COLUMN=8
5240 N=R(1)
5250 GOSUB 8000:REM Update X register display
5260 GOTO 11500:REM Get next instruction
5490 REM ** Mark and update the current locn.
5500 ROW=INT(P/10)+1:REM F.PBasic only
5510 COLUMN=(P-10*ROW)*3+31
5520 GOSUB 6000:REM Put the cursor there
5530 PRINT "  ";:REM <2 SPC>
```

```
5540 ROW=12
5550 COLUMN=15
5560 N=P
5570 K=I:REM Save instruction code
5580 GOSUB 8000:REM Update program    counter
5590 N=M(P)
5600 I=P
5610 GOSUB 16500:REM Redraw curent    location
5620 GOSUB 3500:REM Poll the keyboard
5630 I=K:REM Restore instruction code
5640 RETURN
5990 REM ** Position cursor at column & row
6000 CURSOR COLUMN,ROW:PRINT "";
6010 RETURN
7990 REM ** Print N at current coordinates
8000 GOSUB 6000
8010 IF N>9 THEN T$=STR$(N):L=LEN(T$):T$=RIGHT$(T$,L-1)
8020 IF N<10 THEN T$=STR$(N)+" ":REM Force two character width
8030 PRINT MID$(T$,1,2);:REM In range 0-99
8040 RETURN
8090 REM ** Print K at current coordinates
8100 GOSUB 6000
8110 IF K>9 THEN T$=STR$(K):L=LEN(T$):T$=RIGHT$(T$,L-1)
8120 IF K<10 THEN T$=STR$(K)+" ":REM Force two character width
8130 PRINT MID$(T$,1,2);:REM In range 0-99
8140 RETURN
8990 REM ** Draw the display
9000 I=0
9010 CLS
9020 GOSUB 9500:REM 2LH,1RH column
9030 FOR ROW=1 TO 10
9040 COLUMN=0
9050 GOSUB 6000:REM Position cursor
9060 PRINT ":"
9070 FOR J=1 TO 9
9080 COLUMN=J*3-2
9090 N=M(I)
9100 GOSUB 8000:REM Print one memory  element
9110 I=I+1
9120 NEXT J:NEXT ROW
9130 ROW=11
9140 COLUMN=0
9150 GOSUB 6000:REM Position cursor
9160 FOR I=0 TO 29
9170 PRINT "-";
9180 NEXT I
9190 ROW=12
9200 GOSUB 6000:REM Position cursor
9210 PRINT "=00).(X=00).(P=00).(Z=N).(C=N)
9220 ROW=13
9230 GOSUB 6000:REM Position cursor
9240 FOR I=0 TO 29
9250 PRINT "-";
9260 NEXT I
9270 ROW=15
9280 GOSUB 6000:REM Cursor on message line
9290 COLUMN=0
9300 ROW=21
9310 GOSUB 6000:REM Cursor on message line
9320 PRINT "? for HELP"
9330 RETURN
9490 REM ** 2*LH,1*RH screen columns
9500 X=0
9510 RESTORE 310
9520 FOR Y=1 TO 13:READ B
9530 VPOKE Y*40+X+&H3C00,B
9540 NEXT
9550 X=X+1
9560 IF X=2 THEN 9580:REM Job done
9570 GOTO 9520:REM Carry on
9580 RETURN:REM 9030
```

```
9990 REM ** Get the user's next command
10000 ROW=14
10010 COLUMN=0
10020 GOSUB 4000:REM Clear prompt line
10030 PRINT "Command";
10040 INPUT"> "; T$
10050 REM ** Force capitals
10060 C2$=""
10070 FOR C=1 TO LEN(T$):C1$=MID$(T$,C,1)
10080 IF C1$>="a" THEN C1$=CHR$(ASC(C1$)-32)
10090 C2$=C2$+C1$:NEXT
10100 T$=RIGHT$(C2$,LEN(T$))
10110 GOSUB 10130
10120 GOTO 10180
10130 ROW=15
10140 GOSUB 4000:REM Clear message line
10150 ROW=16
10160 GOSUB 4000:REM Clear extra line
10170 RETURN
10180 IF T$="RUN"  THEN 11000
10190 IF T$="SAVE" THEN 12000
10200 IF T$="LOAD" THEN 13000
10210 IF T$="?" THEN 14200
10220 IF T$="QUIT" THEN 10380
10230 IF T$="CLEAR"  THEN 10400
10240 IF T$="STORE" THEN 16000
10270 PRINT "* ";T$;" is not a valid command";
10280 GOTO 10000
10290 REM ** Affirm QUIT/CLEAR command
10300 COLUMN=0
10310 ROW=15
10320 GOSUB 4000
10330 PRINT "Discard current workspace (Y/N)";
10340 INPUT T$
10350 IF T$="Y" THEN RETURN
10360 IF T$="N" THEN 12260
10370 IF T$<>"Y"OR T$<>"N" THEN 10350
10380 GOSUB 10300:REM Print prompt
10390 GOTO 15000:PRINT MENU
10400 GOSUB 10300:REM Print prompt
10410 CURSOR 0,14:PRINT "Working:"
10420 COLUMN=0
10430 ROW=15
10440 GOSUB 4000
10450 I=0
10460 FOR ROW=1TO10
10470 FOR J=1TO10
10480 IF I=99 THEN 10510:REM Last locn
10490 IF M(I)<>0 THEN 10540
10500 IF M(I)=0THEN 10570:REM Carry on
10510 COLUMN=0
10520 GOSUB 9190:REM Clear registers
10530 GOTO 1020:REM Clear variables
10540 COLUMN=J*3-2
10550 N=0
10560 GOSUB 8000:REM Draw at mem. locn
10570 I=I+1
10580 NEXT J:NEXT ROW
10590 REM ** Program has been stopped
10600 ROW=15
10610 COLUMN=0
10620 GOSUB 6000:REM Prepare for message
10630 PRINT "* Program stopped";
10640 GOTO 10000:REM Get a command
10990 REM ** "RUN" command pre-processor
11000 COLUMN=0
11010 ROW=14
11020 GOSUB 4000:REM Clear the prompt line
11030 PRINT "Start address";
11040 GOSUB 4500:REM Get the start of the program
11050 IF N>99 THEN 10000:REM Error
```

```
11060 P=N:REM Set program counter
11070 ABRT=0:REM Clear the abort flag
11490 REM ** "RUN" main loop for each instruction
11500 I=M(P):REM Get next instruction
11510 GOSUB 5500:REM Update display,  check keys
11520 IF ABRT=1 THEN 10600:REM Quit if requested
11530 COLUMN=0
11540 ROW=15
11550 GOSUB 6000:REM Put cursor on   message line
11560 IF I<1 THEN 11580:REM Halt code
11570 IF I<=MAX THEN 11620:REM Other instruction
11580 IF I=0 THEN PRINT "HALT";
11590 IF I<>0 THEN PRINT "* Unknown instruction";
11600 PRINT " at";P;
11610 GOTO 10000:REM Get next command
11620 IF P<>99 THEN 11650:REM Not end of memory
11630 PRINT "* No end on program";
11640 GOTO 10000:REM Get next command
11650 P=P+1
11660 J=M(P):REM Get operand
11670 P=P+1:REM Point to next instruction
11680 K=0:REM Assume a register A instruction
11700 IF I>10 THEN IF I<17 THEN K=1:REM Wrong ! Register X
11710 ON I GOTO 20000,20100,20200,20300,20400,20500,20600,20700,20800,20900
0,20100,20200,20300,20400,20500,21000,21100,21200:REM Execute instructions
11990 REM ** "SAVE" current program
12000 GOSUB 12500:REM Get Filename
12010 POKE&H9808,PEEK(&H8162):POKE&H9809,PEEK(&H8163)
12020 POKE&H980A,PEEK(&H8166):POKE&H980B,PEEK(&H8167)
12030 COLUMN=0
12040 ROW=15
12050 GOSUB 4000:REMClear message line
12060 PRINT "CONTINUE SAVE (Y/N)";
12070 INPUT T$
12080 IF T$="Y" THEN 12110
12090 IF T$="N" THEN 12260
12100 IF T$<>"Y"OR T$<>"N" THEN 12080
12110 COLUMN=0
12120 ROW=15
12130 GOSUB 4000:REMClear message line
12140 PRINT "Press SAVE & LOAD then <CR>";
12150 INPUT "*";T$
12160 COLUMN=0
12170 ROW=15
12180 GOSUB 4000:REMClear message line
12190 PRINT "Saving ";FL$
12200 CALL&H9814
12210 CURSOR0,17:PRINT "             "
12220 CURSOR0,18:PRINT "             "
12230 CURSOR0,19:PRINT "             "
12240 CURSOR0,20:PRINT "             "
12250 CURSOR0,15:PRINT "Saving end   "
12260 GOSUB 10130:REM Clear extra line
12270 GOTO 10000:REM Get next command
12490 REM ** Get filename
12500 ROW=14
12510 COLUMN=0
12520 GOSUB 4000
12530 PRINT "Enter Filename * ";
12540 INPUT"EASY.";T$
12550 FL$="EASY."+T$
12560 FOR BZ=LEN(FL$) TO 16:REM Pad  filename with blanks
12570 FL$=FL$+CHR$(32):NEXT
12580 RETURN
12990 REM ** 'LOAD' memory from tape
13000 COLUMN=0
13010 ROW=15
13020 GOSUB 4000
13030 PRINT "Destroy existing data (Y/N)";
13040 INPUT T$
13050 IF T$="Y" THEN 13080
```

```
13060 IF T$="N" THEN 12080
13070 IF T$<>"Y"OR T$<>"N" THEN 13050
13080 COLUMN=0
13090 ROW=15
13100 GOSUB 4000
13110 COLUMN=0
13120 ROW=14
13130 GOSUB 6000
13140 PRINT "Working:"
13150 POKE&H9808,PEEK(&H8162):POKE&H9809,PEEK(&H8163)
13160 POKE&H82A2,0
13170 CALL&H9844
13180 CURSOR 0,17:PRINT "                              "
13190 CURSOR 0,16:PRINT " Loading end  "
13200 I=0
13210 FOR ROW=1TO10
13220 FOR J=1TO10
13230 IF I=99 THEN 13260
13240 IF M(I)<>0 THEN 13290
13250 IF M(I)=0 THEN 13320
13260 COLUMN=0
13270 GOSUB 9190
13280 GOTO 10000
13290 COLUMN=J*3-2
13300 N=M(I)
13310 GOSUB 8000
13320 I=I+1
13330 NEXT J:NEXT ROW:P=0
13490 REM ** Tape routine Mcode loader
13500 RESTORE 60
13510 FOR X=&H9814 TO &H987D
13520 READ A$:POKEX,VAL("&H"+A$):NEXT
13530 RETURN:REM to 15620
13990 REM ** 'HELP' command received
14000 SCREEN 2,1:REM Change the screen
14010 COLOR 1,3,(0,0)-(255,191),3
14020 PRINT"     Valid EASYCODE commands are:
14030 PRINT
14040 PRINT"     STORE to enter data on program"
14050 PRINT"     CLEAR to reset MON+ memory"
14060 PRINT"     RUN   to execute a MON+ program"
14070 PRINT"     SAVE  to store one on tape"
14080 PRINT"     LOAD  to read one from tape"
14090 PRINT"      ?    to view this message"
14100 PRINT"     QUIT  to return to Basic":PRINT
14150 CURSOR 20,160
14160 PRINT"       Loading Working Screen"
14170 IF SD=0 THEN 14240
14180 CURSOR 10,160:PRINT CHR$(5)
14190 CURSOR 10,160:PRINT"       Press space bar to continue";
14200 SCREEN 2,2
14210 IF INKEY$<>CHR$(32)THEN 14210
14220 CURSOR 0,15:PRINT "             "
14230 GOTO 14300
14240 SCREEN 1,1
14250 GOSUB 15650:REM Display menu?
14260 SD=SD+1:SCREEN 1,2
14270 GOSUB 9000:REM Redraw display
14280 SCREEN 2,2
14290 GOTO 14180
14300 SCREEN 1,1
14310 CURSOR 0,15:PRINT "             "
14320 GOTO 10000:REM Get next command
14990 REM ** 'QUIT' routine (nice and simple!)
15000 CLS
15010 END:REM That's all folks
15490 REM ** Opening screen
15500 SCREEN 1,1:CLS
15510 CURSOR 10,0:PRINT "E A S Y C O D E"
15520 CURSOR 13,4:PRINT "Simulated"
15530 CURSOR 11,5:PRINT "Machine  code"
```

```
15540 CURSOR 14,6:PRINT "Monitor"
15550 CURSOR 8,9:PRINT "for 16k (level 111A)"
15560 CURSOR 12,11:PRINT "SEGA SC3000"
15570 CURSOR 3,15:PRINT "Copyright (C) 1984 Simon Goodwin."
15580 CURSOR 3,16:PRINT "SEGA  version 1985 David Coursey."
15590 IF TR>0 THEN 15630
15600 CURSOR 9,19:PRINT "Loading Mcode data"
15610 GOSUB 13500
15612 CLS:PRINT "Before EASYCODE can be run, you must  delete the following line
s then SAVE  the program as your working copy."
15614 PRINT:PRINT " 50-180, 13490-13530 & 15590-15616."
15616 END
15620 TR=TR+1
15630 CURSOR 5,19:PRINT "       Loading Menu
15640 RETURN:REM to 1020
15650 CURSOR 5,19:PRINT "Press space bar to continue"
15660 IF INKEY$<>CHR$(32) THEN 15660
15670 RETURN:REM 14260
15990 REM ** 'STORE' data or program
16000 COLUMN=0
16010 ROW=15
16020 GOSUB 4000:REM Clear messages   (for later)
16030 ROW=14
16040 GOSUB 4000:REM Clear prompt line
16050 PRINT "Enter address (100 to stop)";
16060 GOSUB 4500:REM Get number
16070 IF N>99 THEN 10000:REM Error
16080 K=N
16090 ROW=15
16100 COLUMN=0
16110 GOSUB 4000:REMSet up next prompt
16120 PRINT "Enter data (100 to stop)";
16130 ROW=14
16140 GOSUB 4000:REM Set up varying   prompt
16150 PRINT"Address";K;"=";
16160 GOSUB 4500:REM Get number
16170 IF N>99 THEN 16000:REM Error
16180 I=K
16190 GOSUB 16500:REM Store in memory & display
16200 K=K+1:REM Select next location
16210 IF K<100 THEN 16090:REM Get more
16220 ROW=15
16230 COLUMN=0
16240 GOSUB 4000:REM Clear old message
16250 PRINT "* End of memory reached";
16260 GOTO 10000:REM Get new command
16490 REM ** Put value N in M() and on screen
16500 M(I)=N
16510 ROW=INT(I/10)+1:REM F.P Basic only
16520 COLUMN=(I-10*ROW)*3+31
16530 GOSUB 8000:REM Print the number
16540 RETURN
19990 REM ** LOAD Register;number
20000 R(K)=J
20010 GOTO 5000
20090 REM ** LOAD Register;memory
20100 R(K)=M(J)
20110 GOTO 5000
20120 I=J
20190 REM ** STORE Register;memory
20200 I=J
20210 N=R(K)
20220 GOSUB 16500:REM Display alteration
20230 GOTO 11500:REM No flags - just  get next
20290 REM ** LOAD Register;Register'
20300 R(K)=R(1-K)
20310 P=P-1:REM Only a 1 char. instruction
20320 GOTO 5000
20390 REM ** ADD Register;number
20400 R(K)=R(K)+J
20410 GOTO 5000
```

```
20490 REM ** SUB Register;number
20500 R(K)=R(K)-J
20510 GOTO 5000
20590 REM ** SUB A;@X
20600 R(0)=R(0)-M(R(1))
20610 P=P-1:REM Only a 1 char. instruction
20620 GOTO 5000
20690 REM ** JUMPNC;address
20700 IF CARRY=0 THEN P=J
20710 GOTO 11500
20790 REM ** JUMPNZ;address
20800 IF ZERO=0 THEN P=J
20810 GOTO 11500
20890 REM ** JUMP;address
20900 P=J
20910 GOTO 11500
20990 REM ** LOAD A;@X
21000 R(0)=M(R(1))
21010 P=P-1:REM 1 char. instruction
21020 GOTO 5000
21090 REM ** STORE A;@X
21100 N=R(0)
21110 I=R(1)
21120 P=P-1:REM 1 char. instruction
21130 GOSUB 16500:REM Store & display
21140 GOTO 11500
21190 REM ** ADD A;@X
21200 R(0)=R(0)+M(R(1))
21210 P=P-1
21220 GOTO 5000
```

EASYCODE

Extra lines to convert 16K to 32K SEGA version.

Merge this with the 16K version.

```
190 REM ** Assembler text & codes
200 DATA STORE A;@X,18,LOAD A;@X,17
210 OATA STORE A;@,-3,STORE X;@,-13
220 OATA LOAD A;@,-2,LOAD A;X,4
230 OATA SUB A;@X,7,LOAD X;@,-12
240 OATA LOAD X;A,14,AOO A;@X,19,JUMPNZ;,-9
250 DATA JUMPNC;,-8,LOAO A;,-1,LOAD X;,-11
260 OATA PUSH A,20,PUSH X,21,RETURN,25
270 OATA ADO A;,-5,AOO X;,-15,SUB A;,-6
280 OATA SUB X;,-16,JUMP;,-10,POP A,22
290 OATA POP X,23,CALL;,-24,HALT,0
300 REM ** Screen display data
310 DATA 32,49,50,51,52,53,54,55,56,57,45,40,45
320 OATA 48,48,48,48,48,48,48,48,48,48,45,65,45
330 OATA 41,41,41,41,41,41,41,41,41,41,45,32,45
1040 MAX=25:REM Highest instruction code
1050 DIM D$(MAX),E(MAX),S(9)
6490 REM Set up data for Assm. & Oism.
6500 RESTORE 190
6510 FOR I=0 TO 25
6520 READ O$(I),E(I):REM Text & instruction No.
6530 NEXT I
6540 RETURN
6990 REM ** Push N onto stack
7000 COLUMM=33
7010 ROW=10-STACK
7020 IF ROW=0 THEN 7110
7030 S(STACK)=M
7040 GOSUB 8000
7050 STACK=STACK+1
7060 COLUMN=34
7070 ROW=12
7080 M=STACK
7090 GOSUB 8000:REM Update S display
7100 RETURN
7110 COLUMN=0
7120 ROW=15
7130 GOSUB 6000:REMPrepare for message
7140 PRINT "* Stack full";
7150 GOTO 11600:REM Leaves 1 GOSUB stacked
7490 REM ** POP N from top of stack
7500 STACK=STACK-1
7510 IF BTACK<0 THEN 7590:REM Whoops, error
7520 GOSUB 7060
7530 COLUMN=33
7540 ROW=10-STACK
7550 GOSUB 6000:REM Prepare to clear  old entry
7560 PRINT "  ";:REM <2 SPC>
7570 N=S(STACK)
7580 RETURN
7590 PRINT "* Mothing left on stack";
7600 GOTO 11600
9020 GOSUB 9500:REM 2LM,1RH column
9120 NEXT J:PRIMT"(":NEXT ROW
9160 FOR I=0 TO 36
9210 PRINT "=00).(X=00).(P=00).(Z=N).(C=N).(S=00)"
9240 FOR I=0 TO 36
9490 REM ** 2*LN,1*RN screen columns
9500 X=0
9510 RESTORE 310
9520 FOR Y=1 TO 13:READ B
9530 VPOKE Y*40+X+&H3C00,B
9540 NEXT
9550 X=X+1:IF X=2 THEN X=39
9560 IF X=40 TNEN 9580:REM Job done
9570 GOTO 9520:REM Carry on
9580 RETURN:REM 9030
10245 IF T$="HCOPY" THEM 22000
10250 IF T$="DIS" THEM 17000
10260 IF T$="ASM" THEN 18000
11080 COLUMM=33:REM Clear stack
11090 FOR ROW=1 TO 10
11100 GOSUB 6000:REM Position cursor
11110 PRIMT "  ";:REM <2 SPC>
11120 NEXT ROW
11130 STACK=0
11140 M=STACK
11150 COLUMM=34
11160 ROW=12
11170 GOSUB 8000:REM Rewrite stack pointer
11180 K=0
11190 GOTO 5000:REM Write A,X etc
11690 IF I=21 OR I=23 THEM K=1
11720 ON I-19 GOTO 21300,21300,21400,21400,21500,21600
14100 PRIMT"      QUIT  to return to Basic"
14110 PRINT"      HCOPY to print working screen":PRINT
14120 PRINT"      DIS   to disassemble memory"
14130 PRINT"      ASH   to assemble into memory"
15550 CURSOR 14,8:PRIMT "for 32k"
16990 REM ** OIssasembler - main loop
```

```
17000 ROW=14
17010 COLUMM=0
17020 GOSUB 4000:REM Clear prompt line
17030 PRINT "Disassemble from";
17040 GOSUB 4500:REM Get number
17050 IF N>99 THEM 10000:REM Whoops
17060 SCREEN 2,1:CLS:SCREEN 2,2:M1=0:M2=0
17070 PRIMT "   Addr...Value....Disassembly";
17080 FOR P=16 TO 112 STEP 8
17090 GOSUB 17500:REM Disassemble 1 line
17100 NEXT P
17110 PRIMT:PRINT:PRINT
17120 PRINT "   Continue disassembly (Y/N)";
17130 IF INKEY$<>"N" THEN 17150
17140 GOTO 17180
17150 IF INKEY$<>"Y" THEN 17130
17160 N=N-2:REM Ensure overlap
17170 CLS:GOTO 17070
17180 IF SO=0 THEN CLS:GOTO 17200
17190 CLS:GOTO 17220
17200 SD=SO+1:SCREEN 1,1
17210 GOSUB 9000:REM Redraw screen
17220 SCREEN 1,1
17230 GOTO 10000:REM Back to command
17490 REM ** Disassemble the code at N
17500 IF N>99 THEN RETURN:REM End of  memory
17510 K=M(M):REM Get 1 char
17520 COLUMN=28
17530 ROW=P
17540 GOSUB 8000:REM Address field
17550 COLUMN=74
17560 GOSUB 8100:REM Value field
17570 J=100
17580 FOR I=0 TO 25
17590 IF K<>ABS(E(I)) THEN 17620
17600 J=I:REM Get instruction text Ho.
17610 I=100:REM Flag end of loop
17620 NEXT I
17630 IF J>25 THEN 17800
17640 IF E(J)<0 THEN 17700:REM 2 Char.instruction
17650 COLUMN=128
17660 GOSUB 6000:REM Instruction field
17670 PRINT O$(J);
17680 N=N+1:REM Select next address
17690 RETURN
17700 M=M+1
17710 IF N>99 THEN 17800
17720 K=M(N)
17730 COLUMH=88
17740 GOSUB 8040
17750 COLUMN=128
17760 GOSUB 6000:REM Instruction field
17770 PRINT O$(J);
17780 PRINT K;
17790 GOTO 17680
17800 COLUMN=128
17810 GOSUB 6000:REM Instruction field
17820 PRINT "Data code:";K;
17830 GOTO 17680:REM Exit
17990 REM ** ASSEMBLE - main loop
18000 ROW=14
18010 COLUMN=0
18020 GOSUB 4000:REM Cursor for prompt
18030 PRINT "Assemble to";
18040 GOSUB 4500:REM Get address
18050 IF N>99 THEN 10000:REM Error
18060 K=N:REM Save start address
18070 ROW=15
18080 GOSUB 4000:REM Cursor to messageline
18090 PRINT "Assembling. Type 100 to stop";
18100 ROW=14
18110 OOSUB 4000
18120 PRINT K;"=";
18130 IMPUT T$
18140 IF T$="?" THEN 19270
18150 IF T$="100" TNEN 18190
18160 GOSUB 18500:REM Assemble 1 line
18170 COLUMN=0:REM Just in case
18180 IF K<100  TNEN 18070:REM Re-prompt
18190 ROW=14
18200 GOSUB 4000
18210 ROW=15
18220 GOSUB 4000
18230 IF K>99 THEN PRINT "* End of memory reached";
18240 GOTO 10000:REM Get new command
18490 REM ** Assemble one line into M(K) from T$
18500 J=100
18510 FOR I=0 TO 25
18520 IF O$(I)<>LEFTS(T$,LEN(O$(I))) THEN 18570
18530 J=E(I):REM Get instruction code
18540 I=LEN(D$(I)):REM Get length of  instruction
18550 T1$=RIGHT$(T$,LEN(T$)-I):REM  Get remainder
18560 I=100:REM Flag end of loop
```

```
18570 NEXT I
18580 IF J<0 TNEM 18760:REM 2 char.   instruction
18590 IF J<26 THEM 18680:REM 1 char.  instruction
18600 T$="* Unknown:"+T$
18610 ROW=15
18620 GOSUB 4000:REM Clear old message
18630 PRINT T$;
18640 FOR I=0 TO 1500
18650 MEXT I
18660 GOSUB 4000:REM Clear message
18670 RETURM:REM Error exit
18680 IF T1$="" TNEN 18710:REM No trailing junk
18690 T$="* Too long:"+T$
18700 GOTO 18610:REM Print message and return
18710 M=J
18720 I=K
18730 GOSUB 16500:REM Update display
18740 K=K+1:REM 1 more location used
18750 RETURM:REM Success return
18760 IF K<99 THEN 18790
18770 T$="* Only 1 memory space left"
18780 GOTO 18610:REM Print message
18790 IF T1$<"0" OR T1$>"99" THEN 18860
18800 I=VAL(T1$):REM Check range 0-99
18810 IF I>99 THEN 18860
18820 M=ABS(J):REMGet instruction code
18830 GOSUB 18720:REM Store N
18840 N=VAL(T1$)
18850 GOTO 18720:REM Store parameter &return
18860 T$="* Incorrect number:"+T1$
18870 GOTO 18610:REM Report error
18990 REN ** NELP for Assembler user
19000 SCREEN 2,1:CLS:M1=0:M2=1
19010 PRINT "         Valid instructions:";
19020 FOR R=0 TO 95 STEP 8
19030 ROW=R+16
19040 COLUMM=40
19050 GOSUB 6000
19060 PRINT O$(R/8);
19070 IF E(R/8)<0 THEN PRINT "nn";
19080 COLUMN=144
19090 GOSUB 6000
19100 PRINT O$(R/8+13);
19110 IF E(R/8+13)<0 TNEN PRINT "nn";
19120 MEXT R
19130 PRINT:PRIMT:PRINT "     nn is a number from 0 to 99"
19140 PRIMT:PRIMT:PRIMT:PRINT "  Press space bar to continue
19150 SCREEM 2,2
19160 IF IMKEY$<>CHR$(32) THEN 19160
19170 IF SO=0 TNEN GOTO 19190
19180 GOTO 19210
19190 SD=SO+1:SCREEN 1,1:CLS
19200 GORUB 9000:REM Redraw screen
19210 SCREEN 1,1
19220 ROW=16
19230 COLUMM=0
19240 GOSUB 4000
19250 COLUMM=0
19260 GOTO 18070
19270 IF M2=1 THEN 19150
19280 CURSOR 0,15:PRINT "Loading Menu                      "
19290 GOTO 19000
21290 REN ** PUSH Register
21300 M=R(K)
21310 GOSUB 7000:REM Put N on stack
21320 P=P-1
21330 GOTO 11500
21390 REM ** POP Register
21400 GOSUB 7500:REM Get N from stack
21410 R(K)=N
21420 P=P-1
21430 GOTO 5000
21490 REM ** CALL;Address
21500 N=P:REM Save current program counter
21510 GOSUB 7000:REM Push address
21520 P=J:REM Start processing there
21530 GOTO 11500
21590 REM ** RETURM
21600 GOSUB 7000:REM Get return address
21610 P=N:REM Start processing there
21620 GOTO 11500
21990 REM ** NCOPY
22000 IM=1
22010 FOR VP=&H3C00 TO &H3E7F
22020 VD=VPEEK(VP)
22030 LPRINT CHR$(18):LPRINT "S1"
22040 LPRINT "P";CHR$(VD);
22050 IF IN=40 THEN GOSUB 22100
22060 IN=IN+1
22070 NEXT
22080 LPRIMT"A":GDTO 10000
22100 LPRIMT CHR$(18):LPRINT "A":LPRIMT CNR$(18):IN=0
22110 RETURM
```

CALCULATING VALUES FOR PARALLEL RESISTORS

B. Pycroft

```
10 REM $ (+> Parallel Hesistors.
20 REM H0=Value wanted`'R2=guess`'R3=exact value required with R2 to give R0
30 REM H4=last guess`'R5,R6,R7=sub-variables
40 REM D1=Last guess exponent;save location`'E1=guess exponent
50 REM A1(1)=Array of std values.`' R1 = Value of given pair.
60 REM  1,J,K = Pointers into array A1(n)
70 REM  Calculations based on equ'ns|-1/Rwanted= 1/Rstd+1/Rexact  ~  {1}
80 REM  1/Rexact= 1/Rstd+1/Rexact {2}
90 REM If (1} gives  tolerance the OK,fine, ELSE (2} calculated & a std      val
ue of Rexact' attempted. Ifthis value is unsuitable, then a new value of Rstd is
 tried in {1}
100 REM Tests are repeated until either the tol'ce is met OR Rstd exceeds     tw
ice the value wanted.
110 REM The last test is because the eg'ns ASYMPTOTICALY approach 2Rwtd, &
120 REM if 2Rwtd is passed, then trials have failed!!
130 REM $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
140 REM  Original program produced for MICRO-BEE by A.Woodfeild.
150 REM -S-E-G-A- version by B.PYCROFT
160 REM =========                 =========
170 REM ``````````````````````````````
180 CLS:PRINT "   PARALLEL RESISTOR CALCULATOR."
190 PRINT "   =========================="
200 PRINT
210 DIMA1(15):REM STD VALUE TABLE
220 FOR I=1 TO 13
230 READ A1(I)
240 NEXT I
250 DATA 1,1.2,1.5,1.8,2.2,2.7,3.3,3.9,4.7,5.6,6.8,8.2,10
260 INPUT "What resistor value do you want (OHMS) ? ";R0
270 INPUT "To what tolerance do you want the parallel value (%) ? ";T0
280 R5=R0:GOSUB 420
290 IF ABS(R0-(R2*10^E1))/R0<=T0/100 THEN GOTO 560
300 R3=ABS((R0*(R2*10^E1))/(R0-(R2*10^E1)))
310 R4=R2:D1=E1
320 K=J
330 R5=R3:GOSUB 420
340 IF ABS(R0-(((R2*10^E1)*(R4*10^D1))/((R2*10^E1)+(R4*10^D1))))/H0<=T0/100 THEN
 GOTO 640
350 IF K=13 THEN 400
360 K=K+1
370 R2=A1(K):E1=D1
380 IF R2*10^E1>2.1*R0 THEN 670
390 GOTO 410
400 K=1:E1=D1+1:R2=A1(K):GOTO 380
410 R3=ABS((R0*(R2*10^E1))/(R0-(R2*10^E1))):R4=R2:D1=E1:GOTO 330
420 E1=0
430 IF R5<10 THEN 470
440 R5=R5/10
450 E1=E1+1
460 GOTO 430
470 FOR I=1 TO 13
480 IF A1(I)>=R5 THEN 500
490 NEXT I
500 R6=AI(I):R7=A1(I-1)
510 IF ABS(R5-R6)<ABS(R5 R7) THEN 540
520 GOTO 550
530 RETURN
540 R2=R6:J=1:GOTO 530
550 R2=R7:J=I-1:GOTO 530
560 PRINT
570 PRINT "A resistor of ";R0;" ⊥ can be made using a single resistor "
580 PRINT "of ";INT(R2*10^E1);" ohms to a tolerance of ";T0; " %":GOSUB 770:GOSU
B 750
590 BEEP:PRINT
600 INPUT "Another run (Y/N) ? ";Z1$
610 IF Z1$="Y" OR Z1$="y" THEN 260
620 BEEP2:PRINT "  Bye-bye":BEEP:BEEP:BEEP
630 END
640 PRINT
650 PRINT "A resistor of ";R0;" ⊥ can be made using 2 std resistors of ";INT(R2*
10^E1);" & ";INT(R4*10^D1);" ohms to a tolerance of ";T0;" %":PRINT :GOSUB 720
660 GOTO 590
670 PRINT
680 PRINT "Sorry, no combination within that tolerance.":GOSUB 750:PRINT
690 INPUT "Want to try another tolerance (Y/N) ? ";Z1$ :BEEP 2
700 IF Z1$="Y" OR Z1$="y" THEN 270
710 GOTO 600
720 R1=(R2*10^E1*R4*10^D1)/(R2*10^E1+R4*10^D1)
730 PRINT "R ⊥ = ";R1
740 PRINT "H % = ";ABS((R1/R0)*-100+100)
750 PRINT "R range = ";INT(R0-((T0/100)*R0));" ~ ";R0;" ~ ";INT(R0+((T0/100)*R0)
)
760 RETUHN
770 PRINT :PHINT "R % = ";ABS((R2*10^E1/R0)*-100+100)
780 RETURN
790 REM < Calc end >
```

SEGA COMPUTER
. . . the final part

B. Brown

INTERESTING BITS AND PIECES.

This chapter is dedicated to all those wives who spend endless hours trying to convince their husbands to give up that stupid toy, and spend more time with them. Gathered together in this chapter are the solutions to a wide range of problems, so now there is no excuse for husbands to spend all night trying all those various programming methods that don't work.

A SEGA PRINT USING STATEMENT:

Some people wish that the SEGA had a PRINT USING statement. Basically this allows you to format numbers which always appear in the same place, and with the same number of decimal places after the decimal point. So here is a routine which will always display numbers to two decimal places, and always place it so that the numbers line up with the decimal point always in the same column.

```
 10 INPUT A
 20 A=INT(A*100+.5)/100
 30 A$=STR$(A)
 40 L=LEN(A$)
 50 FOR I=1 TO L
 60 IF MID$(A$,I,1)="." THEN GOTO 100
 70 NEXT I
 80 A$=A$+".00"
 90 GOTO 110
100 IF I=L-1 THEN A$=A$+"0"
110 FOR K=1 TO 10-I
120 A$=" "+A$
130 NEXT K
140 PRINT A$
150 GOTO 10
```

The value of 10 in line 110 has been used to give a number with twelve characters long. The program would be used as a subroutine within your particular program, and accessed by a gosub statement.

A FAULTY RENUMBERER:

Not that you would want one anyway! No, just a note to say that the SEGA RENUM command does not work properly. To illustrate its major weakness, type in the following program,

```
 10 INPUT" String";A$
 20 IF LEN(A$)<7 THEN GOTO 500
 30 IF LEN(A$)>6 THEN 600
 40 GOTO 10
500 PRINT " A$<7": GOTO 10
600 PRINT " A$>6": GOTO 10
```

Then use the RENUM command. The program will be renumbered as follows,

```
10 INPUT" String";A$
20 IF LEN(A$)<7 THEN GOTO 500
30 IF LEN(A$)>6 THEN 600
40 GOTO 10
50 PRINT " A$<7": GOTO 10
60 PRINT " A$>6": GOTO 10
```

Notice that the line numbers in lines 20 and 30 have not been changed. Whenever a goto or line number follows a string manipulation, the renum feature will not work properly.

ERASING CHARACTERS ON THE GRAPHICS SCREEN:

Try the following program,

```
10 SCREEN 2,2:CLS
20 FOR X=1000 TO 1050
30 CURSOR 150,0:PRINT " Score:";X
40 NEXT
50 END
```

As you will have noticed, the characters written tend to overwrite each other. After a couple of prints, you can't read the score at all. The way to overcome this is by using a print CHR$(5) command. This erases everything to the right of the current cursor position. Modify the program to that below,

```
10 SCREEN 2,2:CLS
20 FOR X=1000 TO 1050
30 CURSOR 150,0:PRINT CHR$(5)
40 CURSOR 150,0:PRINT " Score:";X
50 NEXT
60 END
```

As you notice now, the print chr$(5) statement allows you print in the same position twice. However, note that the chr$(5) erases all information to the right of the cursor (except sprites). Its use must therefore be limited to close to the right hand edge, ie for displaying scores, etc, otherwise it could erase part of your pictures or graphic displays.

SOME NOTES ABOUT THE GRAPHICS:

There appear to be some strange things happening when using the graphics screen. This is due to the routines in ROM being designed with circles etc in mind. An example of this limitation follows,

```
10 SCREEN 2,2: CLS : COLOR 1,11,
   (0,0) - (255,191),12
20 LINE (57,50) - (100,100),15,BF
30 CURSOR 66,75: COLOR 1,4
40 PRINT "test"
50 GOTO 50
```

As you probably guessed, "test" is not printed and the background color is ignored. This is because the routine does not erase the previous contents of the video screen when writing new data to it. A possible solution is to add these lines to the previous program,

```
 5 ZX=&H2000: ZC=&H14
25 GOSUB 100
45 GOSUB 110
```

```
100 FOR Y=70 TO 90:BLINE(64,Y)-
    (95,Y): NEXT: RETURN
110 FOR X=64 TO 95 STEP 8
120 FOR Y=70 TO 90
130 VPOKE INT(Y/8)*256+INT(X/8)*8
    +YMOD8+ZX,ZC
140 NEXT: NEXT: RETURN
```

This demonstrates the writing to the color attribute area of the graphics screen. This technique should be used to add more color onto the screen, as the graphic chip does allow 16 colors to be used in a character block (ie 8 x 8). The computer is capable of generating color displays rivalling most computers today, and should be comparable to more expensive computers if programmed correctly.

LISTING PROGRAMS:

When listing Basic programs, pressing the SPACEBAR will pause the listing. Pressing it again, the listing will continue.

HALTING THE GAMES CARTRIDGES:

Pressing RESET will halt the game, while a further press will restart the game.

LOAD OR SAVE VARIABLES, MACHINE-CODE PROGRAMS, STRING ARRAYS ETC:

Well, we may as well go for broke on the last topic in this book. If you have survived to this point then congratulations are in order! By now, some of the concepts should be clicking together and so to finally put you off the deep end, lets get into saving or loading variables etc.
Basic Principle involved: We have already discovered that Basic uses locations in the Reserved RAM area in order to locate where to find the program, variables, strings etc. The LOAD and SAVE routines look up locations &H8160 to &H8165. These locations store the start and end address's of the Basic program and Variable storage areas. The area of memory between the start and end address of the Basic program is saved to tape, but the storage area isn't. In a flash, we discover that if we replace these start and end address's of the Basic program with the address's for the variables, then call the SAVE routine, the computer will save the variables to tape for us. Having saved them to tape, if we reset the address's to what they were previously, all will be fine, and our program will continue on as usual. The same principle applies to the LOAD process. Okay, so the steps involved in designing this are,

1) Set up a machine-code routine to accomplish the task
2) Save the start/end address's somewhere safe
3) Get the variable address's and put them into where the start/end address's of the Basic program are stored
4) Call the LOAD or SAVE routine in ROM
5) Reset the original address's
6) Return back to Basic ....

Setting up the mcode routine. Lets hide the machine-code in a REM statement.

```
5 REM AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA
```

Line 5 has as many 'A's as possible, about 250 of them. Now the first 'A' in line five is stored at address &H9808. Our machine code routine can thus be poked into address &H9808 onwards (though the length of our routine cannot exceed 250). The pointers that we pick up from locations &H8160' must be saved somewhere safe, so we will store them as follows,

&H9808/9	Poke this with start address to be saved
&H980A/B	Poke this with end address to be saved
&H980C/D	Store &H8160/1 here
&H980E/F	Store &H8162/3 here
&H9810/1	Store &H8164/5 here
&H9812/3	Store &H8166/7 here
&H9814"	Machine code routine

The actual mcode routine written in machine code,

```
ENTRY   LD HL,(8160)
SAVE    LD (980C),HL        ;save Basic start
        LD HL,(8162)
        LD (980E),HL        ;save Basic end
        LD HL,(9808)
        LD (8160),HL        ;new start
        LD HL,(980A)
        LD (8162),HL        ;new end
        CALL 7A69           ;call save routine
        LD HL,(980C)
        LD (8160),HL        ;restore Basic start
        LD HL,(980E)
        LD (8162),HL        ;restore Basic end
        RET

ENTRY   LD A,00
LOAD    LD (82A2),A         ;zero filefound flag
        LD HL,(8160)
        LD (980C),HL        ;save Basic start
        LD HL,(8162)
        LD (980E),HL        ;save Basic end
        LD HL,(8164)
        LD (9818),HL        ;save string start
        LD HL,(8166)
        LD (9812),HL        ;save string end
        LD HL,(9808)
        LD (8160),HL        ;new start
        LD HL,(980A)
        LD (8162),HL        ;new end
        CALL 78FF           ;call load routine
        LD HL,(980C)
        LD (8160),HL        ;restore Basic start
        LD HL,(980E)
        LD (8162),HL        ;restore Basic end
        LD HL,(9810)
        LD (8164),HL        ;restore string start
        LD HL,(9812)
        LD (8166),HL        ;restore string end
        RET                 ;return to Basic
```

The LOAD part of the routine is slightly different, because the string pointers are altered by the load routine. Thus they are saved, and later restored after the load has executed. Location 82A2 is stored with zero this tells the load routine to load the first file it encounters.

The machinecode is now converted to DATA statements, and poked into the 'A's that make up line 5, eg

```
FOR X=&H9808 TO &H9808+number of data bytes
READ A:POKE X,A: NEXT
```

Once this is achieved, the routines can be called and executed. This has been used in the following three programs written by the author,

ACCOUNTS RECEIVABLE
ACCOUNTS PAYABLE
MAILING LIST

USER GROUPS

AUCKLAND (NTH SHORE) SEGA USER GROUP
Contact. Norman Raynel
78 Anzac St
Takapuna
AUCKLAND
Ph 496-841

PUKEKOHE SEGA USER GROUP
Contact· Selwyn Easton
4 Roose Ave
PUKEKOHE
Ph: 86-583

SOUTH COROMANDEL SEGA USER GROUP
Contact· Sid Hawken
PO Box 183
WHANGAMATA
Ph: 58-775

HAMILTON SEGA USER GROUP
PO Box 1548 Hamilton
President· Mr Colin Bell Ph. 73-826
Secretary: Mrs Anne Thrush Ph: 437-312
Meetings held 2nd & 4th Monday of each month at Whitiora Community Centre. Willoughby St at 7.30pm.

TOKOROA SEGA USER GROUP
Contact. Geoff Crawford
1 Pio Pio Pl
TOKOROA
Ph: 67-105

ROTORUA SEGA USER GROUP
Contact: Terry Cole
PO Box 7140
Te Ngae
ROTORUA
Ph 59-325 AH

WHAKATANE SEGA USER GROUP
Contact. K Nightingale
240 King St
WHAKATANE
Ph 84-500

NAPIER SEGA USER GROUP
Contact· Reid Duncan
Store Manager
Agnews Refrigeration
Ph: 55-857. 57-431

HAWERA SEGA USER GROUP
Contact. D M Beale
7A Clive St
HAWERA
Ph 85-108

MARTON SEGA USER GROUP
Contact: Mr H R Miller
41 Alexandra St
MARTON

WELLINGTON SEGA USER GROUP
Contact. Shaun Parsons
PO Box 1871
WELLINGTON
Ph: 897-095 (AH) 727-666 (BUS)

CHRISTCHURCH SEGA USER GROUP
Contact: James O'Donnell
15 Jebson St
Shirley
CHRISTCHURCH
Ph 856-884

TIMARU SEGA USER GROUP
Contact: John Oliver
South Canterbury Computer User Group
PO Box 73
TIMARU
Ph 26-300

OAMARU SEGA USER GROUP
Contact: Bill Dowman
99 Aln St
OAMARU
Ph: 46-250

DUNEDIN SEGA USER GROUP
Central City Computer Interest Group
Box 5260
Moray Place
DUNEDIN
Contact: Graeme Simpson,
Saddle Hill, R D 1, Dunedin.
Ph: (089) 6374

WANTED

Copies of:

Geography 3 and Sprite Generator (not Sprite Editor)
Contact: Mr J.E. Hedges, 87B White St, Rangiora.

DISK DRIVE CONNECTION

For those Sega owners who purchased a disk drive from Farmers Trading without the Computer Connection.
Contact Warwick, Phone 444-9081

HI-SCORE CHALLENGE

Borderline	16,802,820	Michael Wilkinson
Congo Bongo	175,490	David Downs
Flicky	2,718,900	B. A. Smaill
Lode Runner	38,000	Steve Biggs
Monaco GP	210,062	Stewart Parkes
Pacar	1,495,500	Tony Sasso
Pop Flamer	117,000	Richard Hendra
Sinbad Mystery	190,000	Stewart Parkes
Star Jacker	398,000	Steve Biggs
Video Flipper	999,880	Andre Stokes
Yamoto	85,000	Stewart Parkes
N-Sub	76,250	Reuban Jackson
Safari Hunting	8,400	David Downs
Safari Race	30,000	Jonathan Fletcher
Orguss	18,500	David Downs
Champion Tennis	9-Love	David Downs

Challenge these Hi-Scores for Cartridge games by sending us yours.

OVERSEAS SEGA USERS CLUBS

Klaus Pinker
P.O. Box 18
Belconnen
ACT 2616
Ph: 062 30 2334

Scott McDonald
2 Coolalie Ave
Camden
NSW 2570
Ph: 046 668 956

John McLennan
65 Highclere Blvd
Marangaroo
WA 6064
Ph: 09 342 3905

Jan Jacobson
10 Pioneer Ave
O'Sullivans Beach
SA 5166
Ph: 08 382 7967

Les Beacall
1/41 Cameron Road
Croydon
VIC 3136
Ph: 03 725 0864

Wayne Ariel
35 Leanne Street
Marsden
QLD 4203

To all Sega Members,

THANKS
FOR EVERYTHING

from

SEGA SOFTWARE SUPPORT